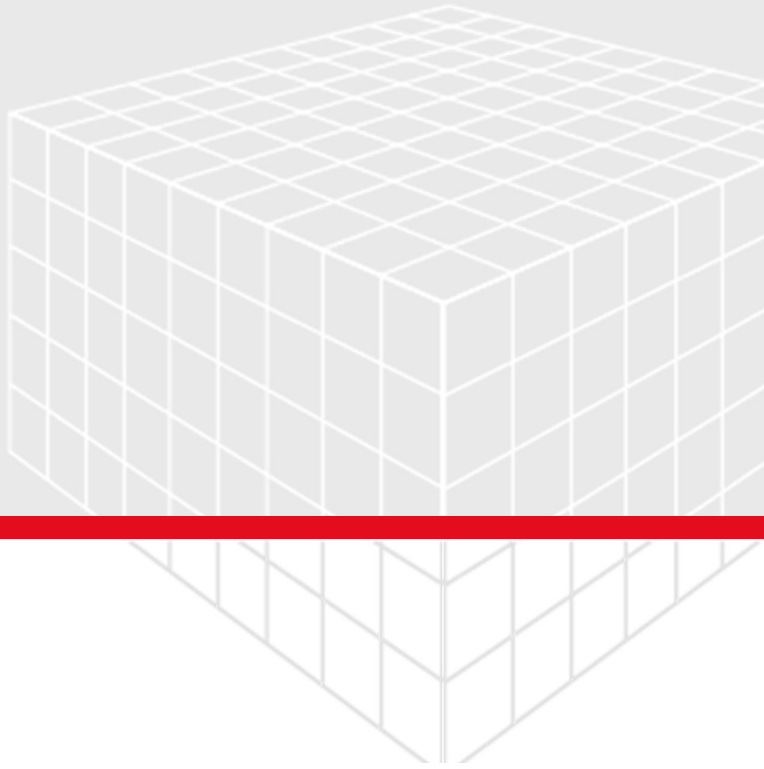
GeoPy Scripting

User Guide



GEODICT Release 2026

Published: July 2026

GEODICT

https://doi.org/10.30423/userguide.geodict

GeoPy Scripting

1. GeoPy Scripting 3

1.1 Definitions 6

1.2 Structure of a GeoPy Macro 7

1.3 Macro Menu 10

1.3.1 Macro Recording 11
1.3.2 Execute Macro / Script 12
1.3.3 Open Scripting Examples 34
1.3.4 Open Scripting Tutorials 34
1.3.5 Session Macro 35
1.3.6 Convert GMC to Python Macro 39
1.3.7 Re-execute Last Python Script 40

1.4 Console and Notifications 41

1.5 Choosing a Text Editor 47

1.6 Parameter Studies 49

1.6.1 Create Parameter Macro 49
1.6.2 Vary Parameters from GeoPy 56
1.6.3 Available Variable Types 58

1.7 Advanced Scripting 66

1.7.1 GeoDict API 66
1.7.2 Shipped Python Modules 129
1.7.3 PowerPoint Report Generation 131
1.7.4 Access to Result Files 137
1.7.5 Create Custom Result Files 142
1.7.6 Access to 3D Result Files 159
1.7.7 Error Reporting 167

1.8 Running from Command Line 173

1 GeoPy Scripting

GeoDict offers the key possibility of recording and executing macros or scripts directly from the GUI (Graphical User Interface) or in the command line.

A **scripting language** is a programming language that automates the execution of tasks which could alternatively be executed one-by-one by a human operator. For this, GeoDict uses Python.

Advantages of automation

In GeoDict, variables and their operations which are defined in a simple Python macro, can be modified using text editor capabilities. The advantages of using macros with variables and other GeoDict macros are:

- Automation of sequences of operations that can run:
  - Without intermediate user interaction.
  - With automatic parameter variation.
- Avoidance of the error-prone and time-consuming process of sequentially introducing values and clicking the same buttons during **frequently repeated processes**.
- Documentation of input parameters providing a record of your activity that can be reproduced by you and by others. All **generation parameters are recorded** in the macro and might be modified at any time.
- Option of **delaying the execution** of the operations listed during the macro recording. Using **Record Only** the macro can be recorded first without actually executing the commands. For example, you record several filtration simulations to run them during the weekend or when cluster time is available. Perhaps you prefer to work on a local computer, but the simulation computations must be done on a remote, more powerful computer.
- Possibility of modifying an isolated parameter in a recorded macro. You can edit the macro with any available text editor (Emacs, WinEdit, WordPad, Notepad, etc.). The modified macro can then be executed.
- Execution of the macro without the intervening GUI, simply as a **command line tool**. For example, when you need to run GeoDict in a batch queue on a Linux cluster or want to control GeoDict by an outside optimization algorithm.
- Variables may take a single value, or multiple values, conveniently defined as a **parameters study** (via a text editor) or in the GeoDict GUI.
- Macros with variables can **reduce the many input parameters** for the various commands in macros to just a few important ones.
- The **relationship between input parameters** may be implemented through arithmetic operations. For example, you chooses the value for the short cross-section diameter of an ellipsoid fiber, and the long one is automatically entered to be 3 times as big.

- Macros with variables can be used to "**program**" GeoDict. For example, when a whole sequence of operations from GrainGeo, ProcessGeo, or LayerGeo is needed to create a realistic geometric model, yet the resolution, porosity, and grain size can vary. Such behavior is seen in the predefined models, e.g., for the GrainGeo module included in the installation folder. In another example, movies may need to be made always with the same corporate color scheme and from the same perspective, on structures of your choice.
- Macros can also be recorded by running GeoDict macros, including parameter studies, to create your own new "effective commands" for GeoDict.

How to get started with automation of your workflows:

- Definitions 6
  View definitions of the most common automation terms in this user guide.
- Structure of a GeoPy Macro 7
  Find out about the structure of a GeoPy macro.
- Macro Menu 10
  Learn how to use the functionality of the macro menu, as for example recording macros.
- Console and Notifications 41
  Use GeoDict's own command-line interface and view notifications.
- Choosing a Text Editor 47
  Choose your preferred text editor to edit GeoPy macros.
- Parameter Studies 49
  The transformation of a simple macro in a parameter macro containing variables is described.
- Advanced Scripting 66
  Dive deeply into GeoPy scripting using many helpful functions.
- Running from Command Line 173
  GeoDict can also be run from the command line.

New in GeoDict 2026

- The macro tutorials 34 and scripting examples 34 are now linked from the macro menu.
- New variable type: gdrstring 61.
- The parameter "Release" is now obligatory in the gd.runCmd 67 API function.

Important! GeoDict 2026 does not effectively support macro syntax older than GeoDict 2021. Errors may occur, when running macros with older syntax or loading parameter settings from them.

Join the conversation

Visit the Community on GeoDict Forum to be inspired and get answers to top questions.
Find illustrative application examples on the website.
Our tutorials and seminar videos give you a first hands-on experience. They are collected in the Learning Center.
Send your Support Request to our technical Support.

1.1 Definitions

The following lists the most important **definitions** of commonly used automation terms for better comprehensibility:

- A **Command** is a directive to a computer program, interpreting to perform the corresponding task.
- In a **Macro** a sequence of commands is saved from the GUI and can be replayed at any time. GeoDict macros can be edited in any available text editor.
- All commands in the GeoDict modules are controlled by **Parameters** that can be edited in the respective module sections. Different parameters lead to different results. These parameters can be recorded in macros, where they can also be edited.
- **Python** is the default interpreted programming language for GeoDict macros.
- **GeoPy** is a short form of GeoDict Python and refers to the programming language of GeoDict macros.
- **GMC** is the old programming language used in GeoDict macros. It can still be used but it is recommended to switch to GeoPy.
- **Command lines** are commands in form of successive lines of text, used in a command-line interface.
- In computer programming **Variables** are used to store information, e.g., in form of numbers (integer, float), text (string) or module parameters (dictionary).

1.2 Structure of a GeoPy Macro

GeoPy (GeoDict Python) macros are scripts running a sequence of commands, even from different licensed modules. Their suffix is .py and they consist of (at least) four blocks:

1. **Header = {}** contains general information with comments on the release, recording time, the recorder or creator and the system used. GeoDict commands refer to the first entry of the header, the 'Release'. Thus, never delete this line.

```
Header = {
  'Release'      : '2026',
  'MinorVersion' : '0',
  'Revision'     : '90640',
  'BuildDate'    : '14 Oct 2025',
  'CreationDate' : '22 Oct 2025',
  'CreationTime' : '17:36:43',
  'Creator'      : 'hilden',
  'Platform'     : '64 bit Windows',
  }
```

2. **Description = ''' '''** is automatically generated and by default it simply describes the GeoDict version used for recording the macro in the given time and date, and the licensee. Edit the description to add information about the macro content. The description is shown in the Macro Execution Control 22.

```
Description = '''
Macro file for GeoDict 2026
recorded at 17:36:43 on 22 Oct 2025
by Support of Math2Market GmbH
'''
```

3. **Variables = {}**. When called from the command line (or first level call), the default values for the variables in the *.py file are used. When called from the GeoDict GUI or from another *.py file (second level call), the default values are ignored. Detailed information can be found in the comment section below the Varibles block and, more detailed, in Available Variable Types 58.

```
Variables = [
#   {
#     'Name'           : 'gd_SVP',
#     'Label'          : 'Solid Volume Percentage',
#     'Type'           : 'double',
#     'Unit'           : '%',
#     'ToolTip'        : 'Solid volume percentage of the created structure.',
#     'BuiltinDefault' : 10.0,
#     'Check'          : 'min0;max100'
#   },
  ]

# Explanations of variables syntax:
####################################
# Name:           mandatory, name of the variable by that it can be addressed in the macro, must not contain white spaces!
# Label:          optional,  appears as text in the GeoDict GUI. If not present, then Name is used also as Label
#                            for type gdrstring, this is ignored and the default label is always shown
# Type:           mandatory, known types are bool, boolgroup, double, uint, int, string, gdrstring, filestring, folderstri
# Unit:           optional,  appears only in GUI (not used to rescale any input parameters automatically)
#                            for type gdrstring,  Unit is ignored
#                            for type filestring, Unit contains the file suffix
#                            for type material,   Unit must be solid, fluid or porous
#                            for type combo,      Unit must contain the possible string-values for the variable separated
#                            for type table,      Unit must be a list of type strings, allowed is "int", "float", "string"
# ToolTip:        optional,  appears in GUI (must be in one line)
# BuiltinDefault: optional,  default value which is used in macro (if not given, defaults to 0 or empty string)
#                            for type table, this should be a python list of entries, left to right, top to bottom, e.g. [
# ColumnHeaders:  optional,  only valid for type table: List of header texts for each table column, e.g. ["Column 1", "Sec
# Check:          optional,  known checks are positive, negative, min, max (checks are separated by semicolon)
# Member:         optional,  defines the member of group type variables. For Labelgroups defined by a list, for combogroup

# Examples of variable usage in GeoDict command argument lists:
#################################################################
# 'SolidVolumePercentage' : (gd_SVP, '%'),                    # Set parameter SolidVolumePercentage value of variable gd_S
# 'ResultFileName'        : f'FilterEfficiency_{gd_SVP}.gdr'  # Make parameter ResultFileName include the string converted

# Frequently used Python syntax:
#####################################################
# print('The solid volume percentage is:', gd_SVP)             # Print to console and log - file
#
#   for x in range(0, 100) : # Use "for" loop statement
#     x_sq = x * x
#
#   if gd_SVP < 0:                                            # Use "if" statement
#     print('error')
```

4. The **command block** contains the commands to be executed by GeoDict. It is explained in detail below 8.

```
64  CreateProjectFolder_args_1 = {
67    gd.runCmd("GeoDict:CreateProjectFolder", CreateProjectFolder_args_1, Header['Release'])
68
69  Preferences_args_1 = {
111   gd.runCmd("GeoDict:Preferences", Preferences_args_1, Header['Release'])
112
113 Create_args_1 = {
264   gd.runCmd("FiberGeo:Create", Create_args_1, Header['Release'])
265
266 LoadGdrFile_args_1 = {
269   gd.runCmd("GeoDictResults:LoadGdrFile", LoadGdrFile_args_1, Header['Release'])
270
271 Dilate_args_1 = {
281   gd.runCmd("ProcessGeo:Dilate", Dilate_args_1, Header['Release'])
282
283 ChangeProjectFolder_args_1 = {
287   gd.runCmd("GeoDict:ChangeProjectFolder", ChangeProjectFolder_args_1, Header['Release'])
```

You can download the macro used as an example in this chapter in a zipped folder.

Download GeoDictMacro.zip

Command Block

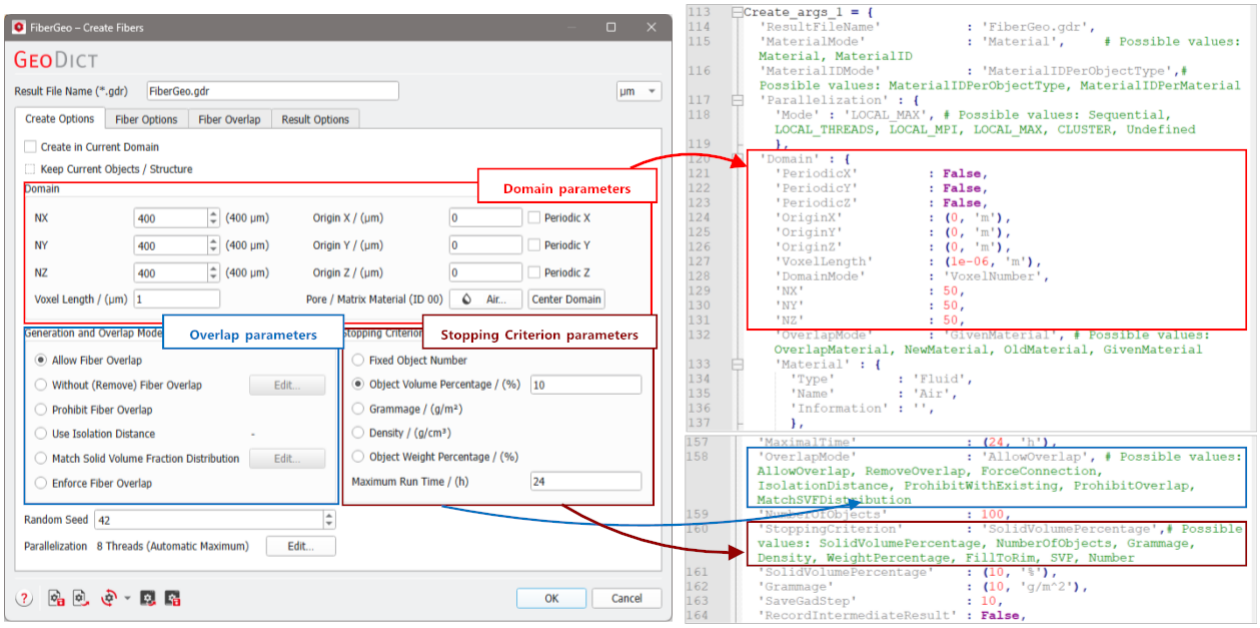
If **Save macro results to new folder** and **Store general preferences in macro** are checked in the **Start Macro Recording** 11 dialog box, the block starts with **GeoDict:CreateProjectFolder** and the **GeoDict:Preferences** which are the settings entered in the settings dialog (**Settings** → **Settings…** in the menu bar).

```
64  CreateProjectFolder_args_1 = {
67    gd.runCmd("GeoDict:CreateProjectFolder", CreateProjectFolder_args_1, Header['Release'])
68
69  Preferences_args_1 = {
111   gd.runCmd("GeoDict:Preferences", Preferences_args_1, Header['Release'])
```

Afterward, the recorded commands can be found. For example, the key **FiberGeo:Create** commands the FiberGeo module to create a structure and to save it as GeoDict structure file (*.gdt).

The command parameters are given in a Python dictionary assigned to a variable called **commandname_args**, e.g., **Create_args** for the FiberGeo Create command. Find all parameters from the GUI in this dictionary given in key : value pairs.

For example, the **'Domain' : {}** parameters in the **Create_args_1** parameters define the periodicity, spatial location (origin), voxel length, and size (NX, NY, NZ) of the structure. After this, the macro continues with the parameters for grammage, overlapping settings, random seed, isolation distance, etc.



Because, in this case, two different materials (both Infinite Circular Fibers) are used in the structure, **'Generator1'** and later **'Generator2'** are called. For these objects the parameter values for 'Material', the 'DiameterDistribution', and the 'OrientationDistribution' of both materials are given.

```
'Generator2' : {
  'Material' : {
    'Type'        : 'Solid',
    'Name'        : 'Glass',
    'Information' : 'Fiber',
    },
  'Probability'             : (0.5, '1'),
  'SpecificWeight'          : (2.58, 'g/cm^3'),
  'Type'                    : 'InfiniteCircularFiberGenerator',
  'UseDTex'                 : False,
  'DiameterDistribution' : {
    'Type'  : 'Constant', # Possible values: Constant, UniformlyInInterval, Gaussian, Table, LogNormal
    'Value' : 6e-06,
    },
  'OrientationDistribution' : {
    'Type'          : 'AnisotropicDirection',# Possible values: Isotropic, AnisotropicDirection, AnisotropicOrientation,
    'DirectionMode' : 'AnisotropyParameter',# Possible values: AnisotropyParameter, DirectionTensor
    'Anisotropy1'   : 5,
    'Anisotropy2'   : 1,
    'Phi'           : 0,
    'Theta'         : 0,
    'Psi'           : 0,
    },
```

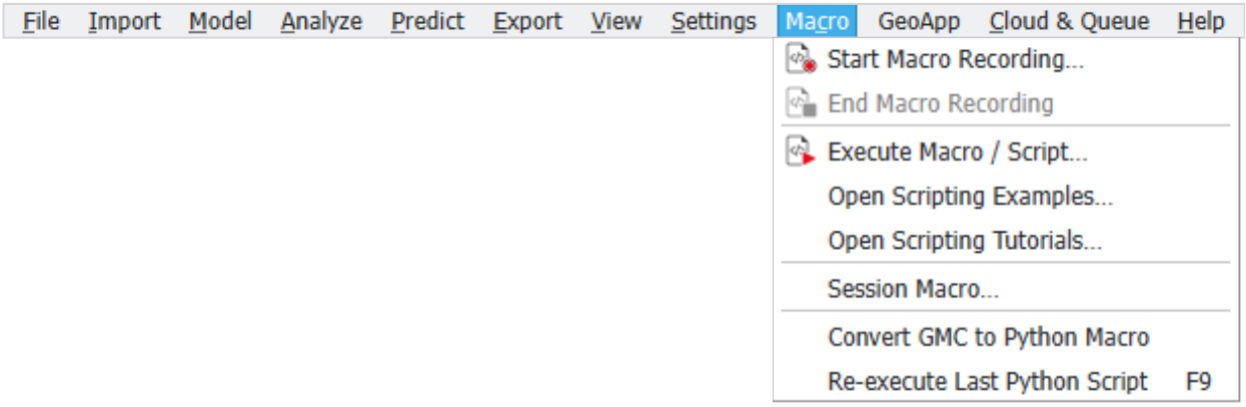
Finally, **gd.runCmd** executes the command. This function is a GeoDict API[67] command and needs three input values:

- GeoDict command, here **FiberGeo:Create**,
- command parameters usually defined above and assigned to a variable, here **Create_args_1**,
- **release year**, usually given by the header[7].

```
gd.runCmd("FiberGeo:Create", Create_args_1, Header['Release'])
```

1.3 Macro Menu

Select **Macro** from the menu bar to access the macro and automation functionality.



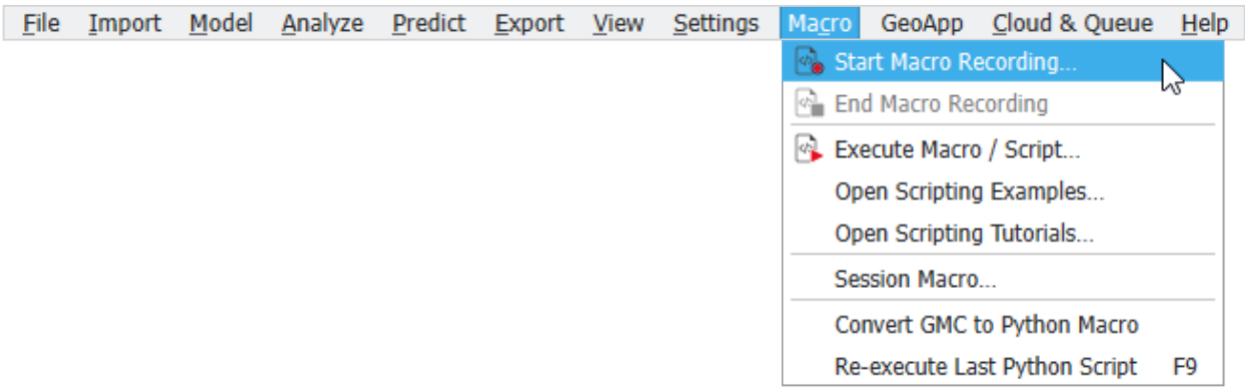
Simple macros are saved while recording 11 a macro or using the Session Macro 35 dialog. A **Simple Macro** only contains the recorded commands from the GUI. A simple macro becomes a Parameter Macro 49 once variables are defined in it. The macro block listing the variables 58 (**Variables = { }**) is already written when a simple macro is recorded, but it is initially empty of variables. Besides defining or editing these variables, you also program the commands for their use.

The Macro menu in the menu bar gives access to the following functionality:

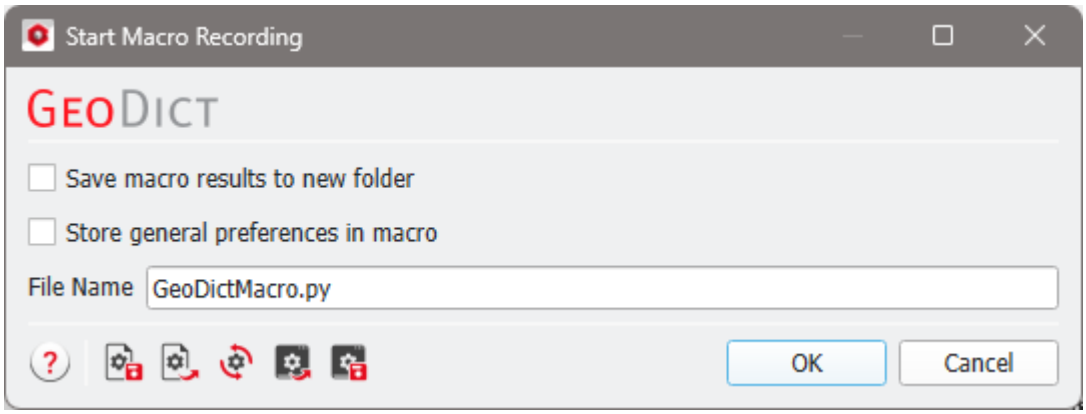
- Macro Recording 11
  Recording a macro while running the different steps from the GUI and end the recording after all needed steps were recorded.
- Execute Macro / Script 12
  Execute a macro or script and access example macros.
- Open Scripting Examples 34
  Find advanced Python scripting examples in the Macro Execution Control.
- Open Scripting Tutorials 34
  Find Python scripting tutorials with detailed descriptions in the Macro Execution Control.
- Session Macro 35
  Observe the commands executed in the current session and record them into a macro using the Session Macro dialog.
- Convert GMC to Python Macro 39
  Convert GMC macros to Python macros
- Re-execute Last Python Script 40
  To quickly execute again the last Python script, select Macro → Re-execute Last Python Script or press F9. The python script is simply executed again without other selections.

1.3.1 Macro Recording

Record a macro while running the steps from the GUI. To begin recording a macro, select **Macro → Start Macro Recording...** in the menu bar.

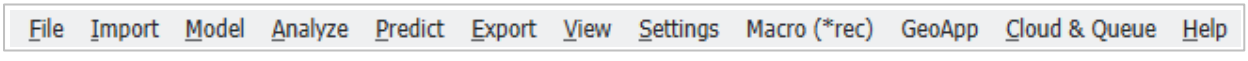


The **Start Macro Recording** dialog opens and offers the following options:

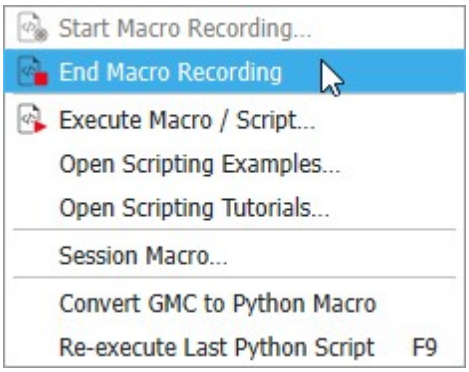


- **Save macro results to new folder** can be selected to include the command **GeoDict:CreateProjectFolder**. The name entered for the macro is given to the newly created project folder. All files created during the execution of the macro are saved in this folder.
- **Store general preferences in macro** can be selected to include the command GeoDict:Preferences in the recorded macro. In GeoDict the preferences can be edited by selecting **Settings → Settings...** from the menu bar.
- At the bottom, enter a **File Name** to save the macro in the project folder.

(*rec) appears to the right of **Macro** in the menu bar as soon as **OK** is clicked.



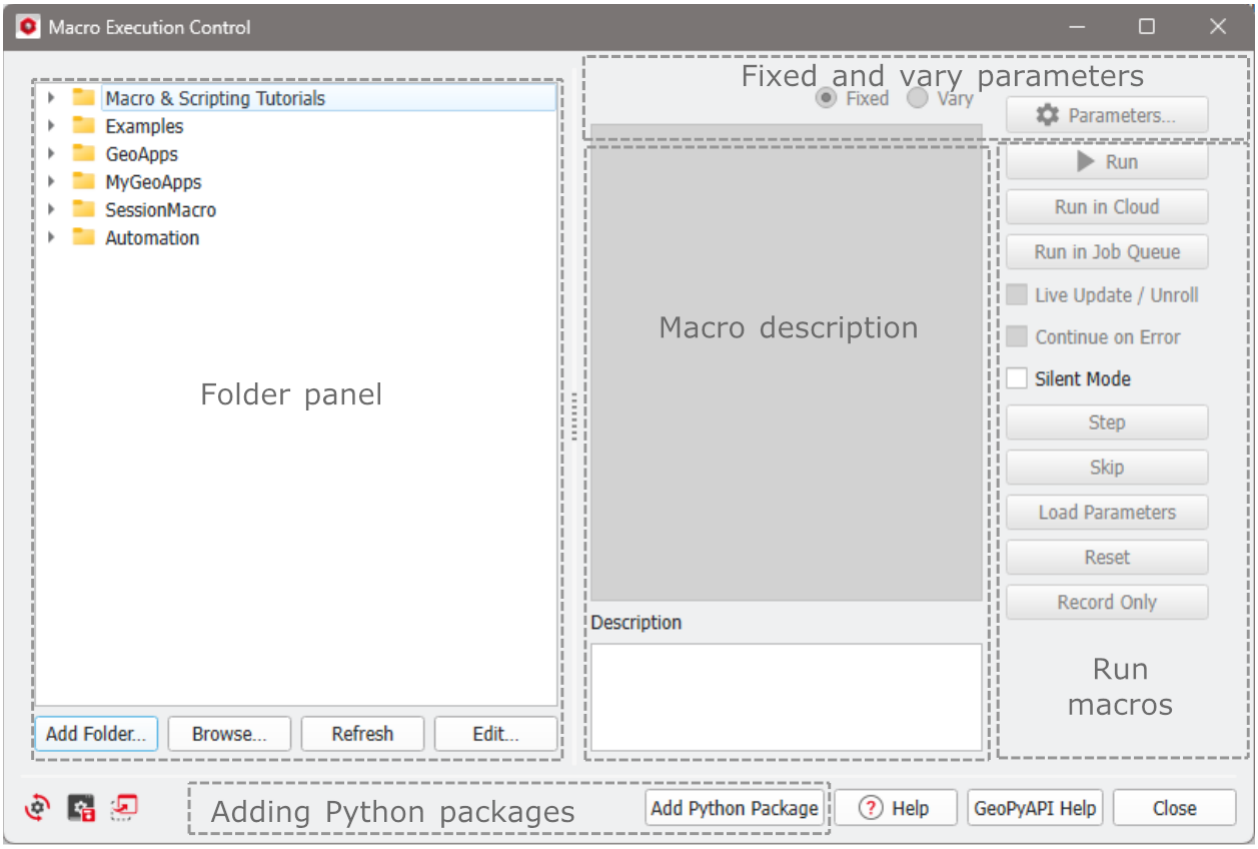
The recording of a macro is stopped by selecting **Macro → End Macro Recording**. This is grayed-out and not selectable unless a macro is being recorded.



1.3.2 Execute Macro / Script

To execute a GeoDict macro or other Python scripts, select **Macro** → **Execute Macro / Script...** to open the **Macro Execution Control** dialog.

The dialog contains two separate parts that can be collapsed and expanded at will by clicking on the dotted line in the middle and dragging it left or right.



Learn to use the Macro Execution Control:

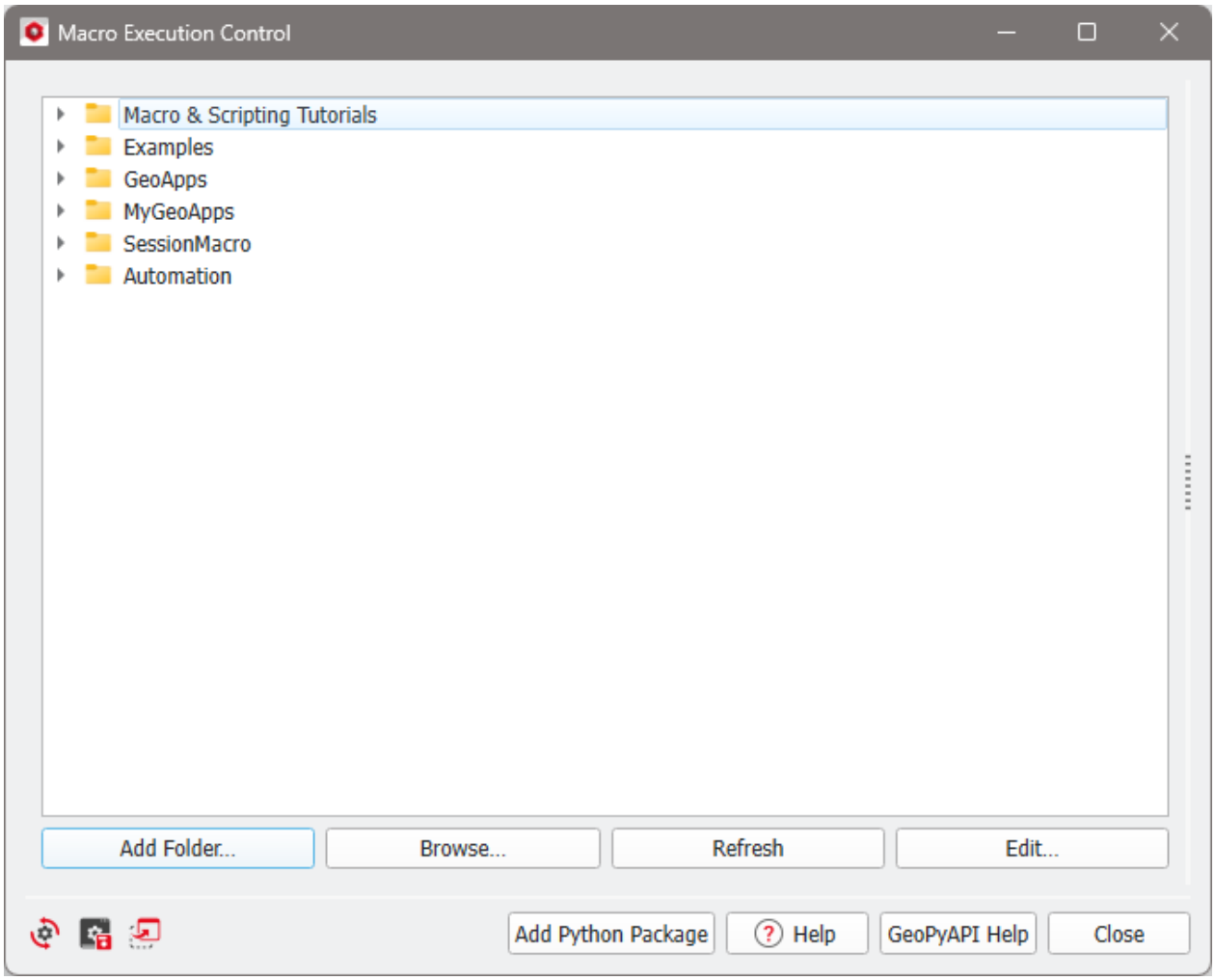
- Folder Panel 13
  Find Python macros in the folder panel.
- Macro Description 20
  Refer to the description on what the macro will do when executed.

Learn to use the Macro Execution Control:

- Fixed and Vary Parameters 22
  Edit the parameters of your macro to run parameter studies.
- Run Macros 29
  Execute the chosen macro.
- Adding Python Packages 32
  Add additional Python packages if needed.
- GeoPyAPI Help 33
  Open a quick overview of all available GeoPy API commands.

Folder Panel

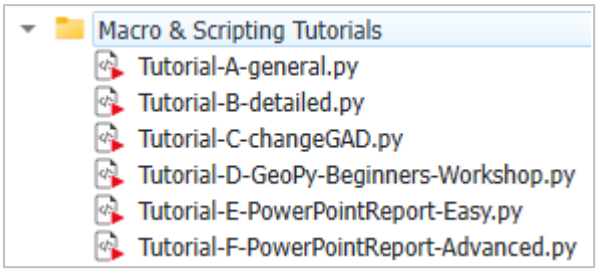
In the left panel of the **Macro Execution Control**, several folders are listed. They contain Python macros that can be executed by clicking Run 29.



Default folders

Macro & Scripting Tutorials

Clicking **Open Scripting Tutorials** in the macro menu or unfolding **Macro & Scripting Tutorials** in the Macro Execution Control shows a list of preinstalled macro tutorials.

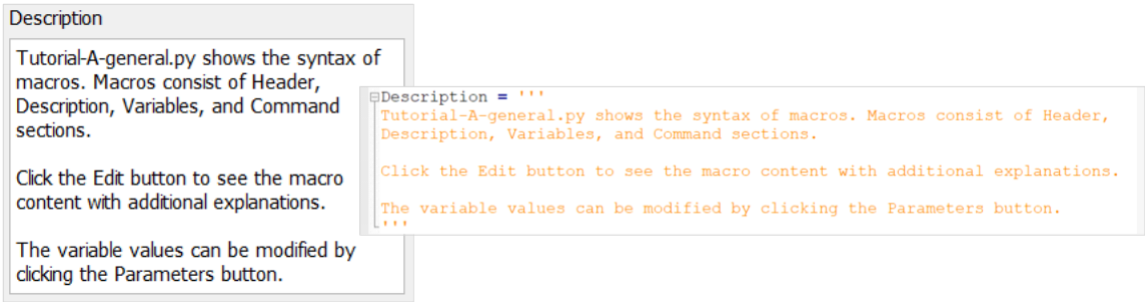


The tutorial macros A, B, C and E need only a GeoDict Base license for execution. The tutorials D and F also need the modules FiberGeo and FlowDict and are the macros created in the workshop videos available on the Math2Market YouTube channel. Find the corresponding links in the macros by opening them in a text editor clicking Edit 13. There, you can also find detailed explanations for all steps.

More advanced example macros can be found in the subfolder Python Scripting Examples 34. These Python scripts also use other GeoDict modules.

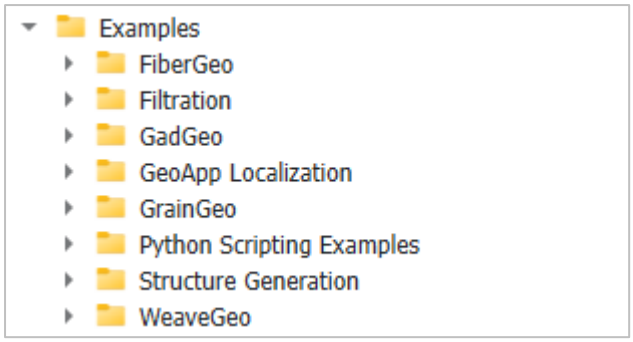
All tutorials have detailed descriptions 20 and thus can be very helpful for getting started with editing Python macros. When selecting one of the available macros, the description area displays a report about the macro. In the macro, this report content can be found between the triple apostrophes after Description = ''' ''', and can be edited at any time after opening the macro with a text editor.

For the Tutorial-A-general.py macro, the text in the macro and in the description area are shown here.



Examples

Clicking **Open Scripting Examples** in the macro menu or directly unfolding **Examples** in the Macro Execution Control gives access to several GeoPy scripts shipped with GeoDict. These scripts are more advanced than the ones to found in the **Macro & Scripting Tutorials** 34 folder. These Python scripts also use other GeoDict modules.

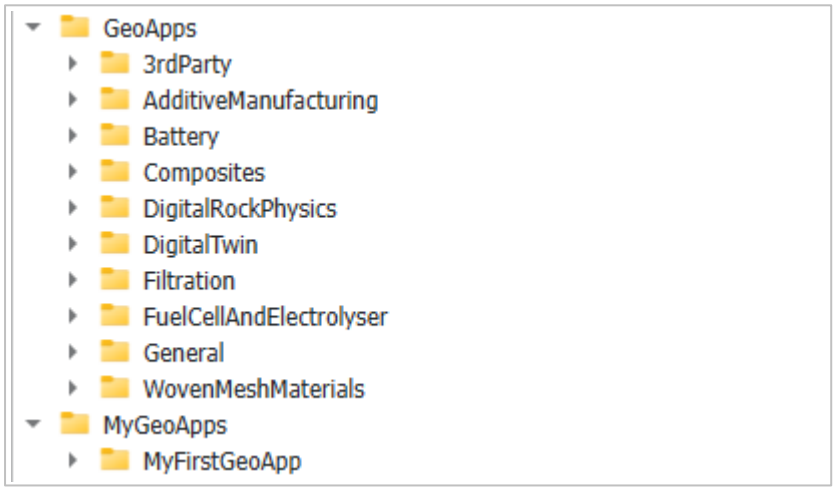


When selecting one of the available macros, the description 20 area displays a report about the macro. In the macro, this report content can be found between the triple apostrophes after Description = ''' ''', and can be edited at any time after opening the macro with a text editor.

Click **Edit** to inspect the macro content in a text editor.

GeoApps and My GeoApps

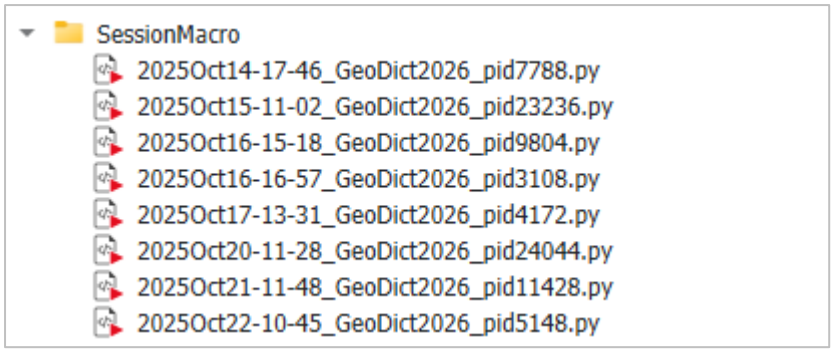
The **GeoApps** folder contains all the GeoApps to be found by selecting **GeoApp** from the menu bar. They are described in the GeoApps chapter.



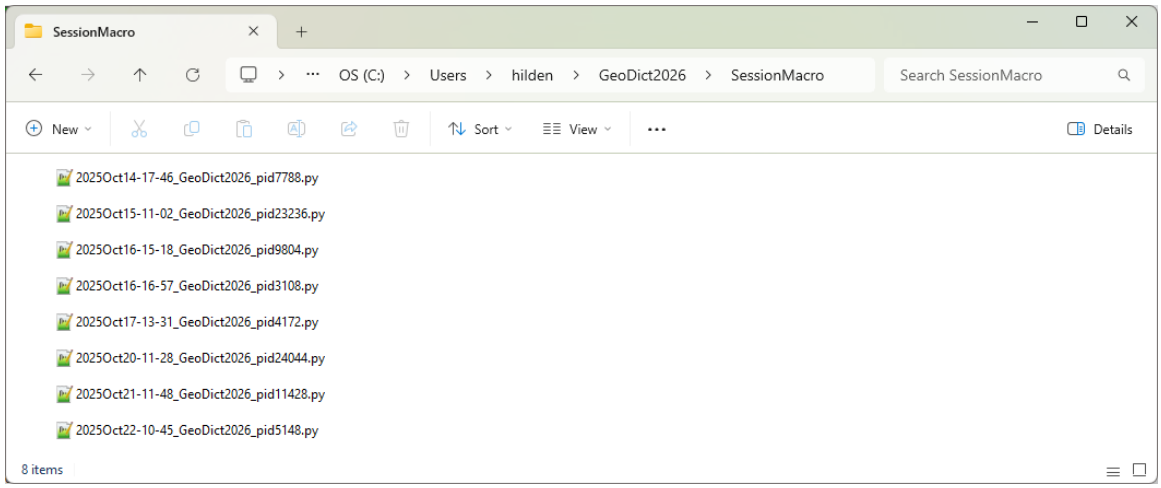
The **MyGeoApps** folder can be filled with your own GeoApps as also described in the GeoApps chapter. By default, it contains **MyFirstGeoApp**, an example GeoApp

Session Macro

In the **SessionMacro** folder, macros containing all commands from the current GeoDict session and the last sessions are saved automatically. The commands contained in these macros are the same that can be found in the **Session Macro** 35 dialog.

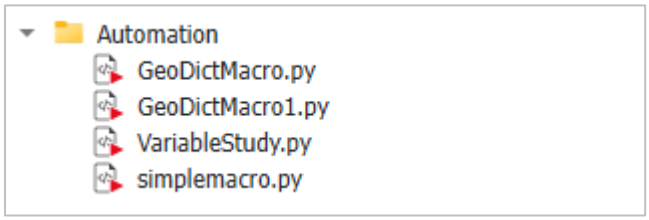


They can be edited at any time since they are saved in the **SessionMacro** folder inside the **GeoDict settings folder**.



Project Folder

The last folder is the selected project folder. All macros inside this folder are shown. Fill this folder for example, by using Record Macro 11 or the Session Macro 35 dialog.



Edit the folder panel

Four buttons are located under the left panel of the **Macro Execution Control** dialog:

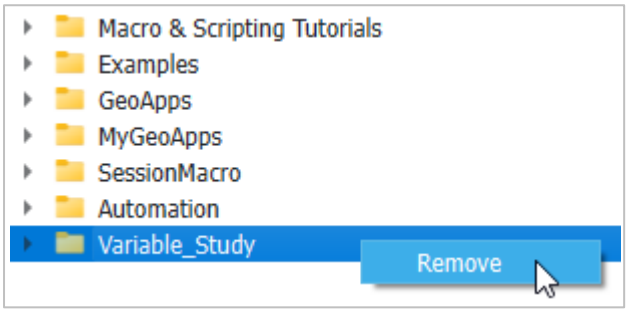
- **Add Folder:** Click to add another folder containing macros to the panel.
- **Browse:** May be used to find and select a macro (*.py, *.gmc) from other than the already listed folders in the left panel. Macros shipped with GeoDict can be found in the default folders 13 in the Macro Execution Control.

- **Refresh:** Clicking Refresh actualizes the list of macros in the pull-down menu. After adding new macros to the project folder, click Refresh to have their file names included in the list.
- **Edit:** GeoDict macros are stored as readable text files and, therefore, can be edited using any text editor, e.g., Editor, WordPad, or Notepad++.

GeoDict does not recognize a file as a macro when the file extension is not *.py or *.gmc. This can happen for example when Windows settings are such that extensions are not shown and, coincidentally the text editor (i.e., Editor or WordPad) automatically adds an extension to the file name (*.txt, *.doc, etc). Then, GeoDict finds **macro1.py.txt** instead of **macro1.py** and does not recognize it as a macro, failing to open it.

The simplest solution is to select a text editor used in programming, e.g., Emacs for Linux systems, or Notepad++ for Windows.

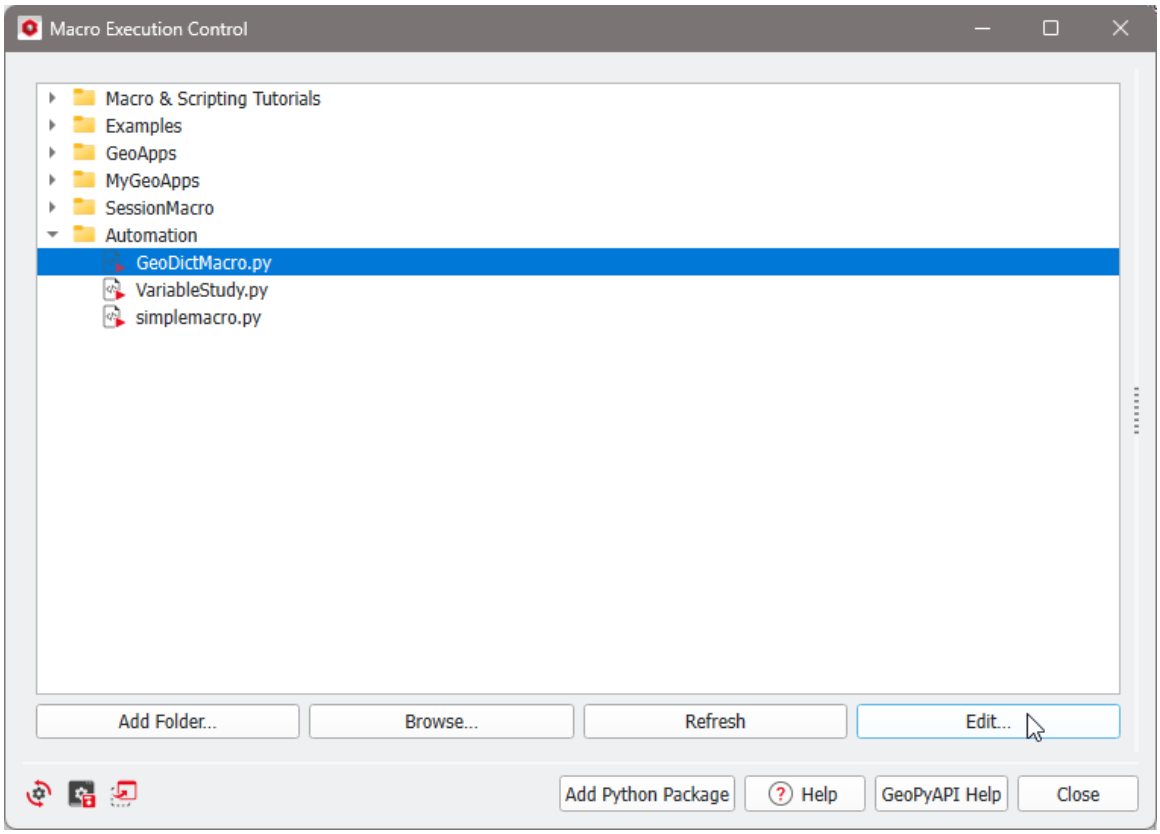
After adding folders with the **Add Folder** or the **Browse** button, they can also be removed at any time by right-clicking on the folder and selecting **Remove**.



Example for editing a macro

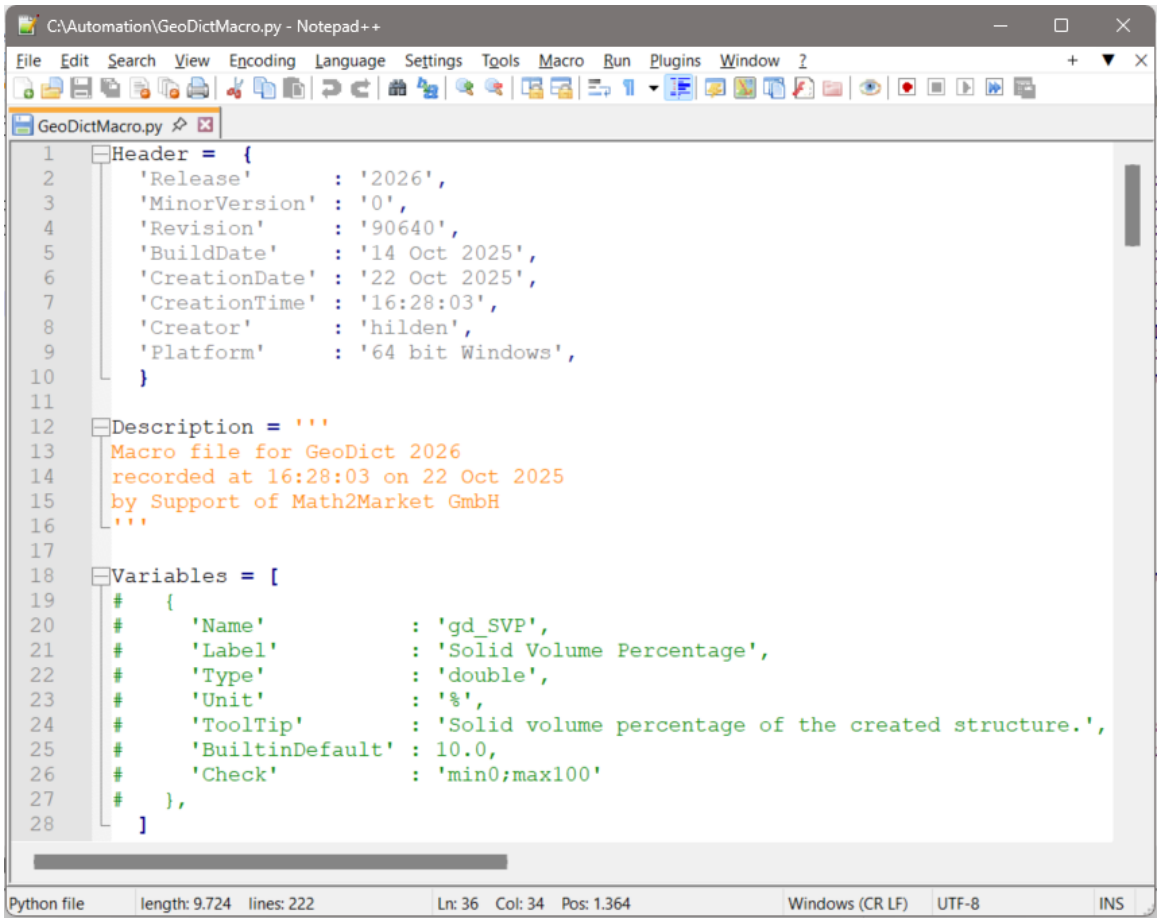
The basic way to edit a macro (e.g., macro.py), is to find the macro file name in the project folder, right click on it, and select **Open With**.... Choose the editor from the list of available programs. However, the macro.py can be directly opened, and then edited from the **Macro Execution Control**. For this, highlight a macro in the left panel.

Click **Edit** to open the selected macro using the designated text editor. The macro then can be examined and edited.

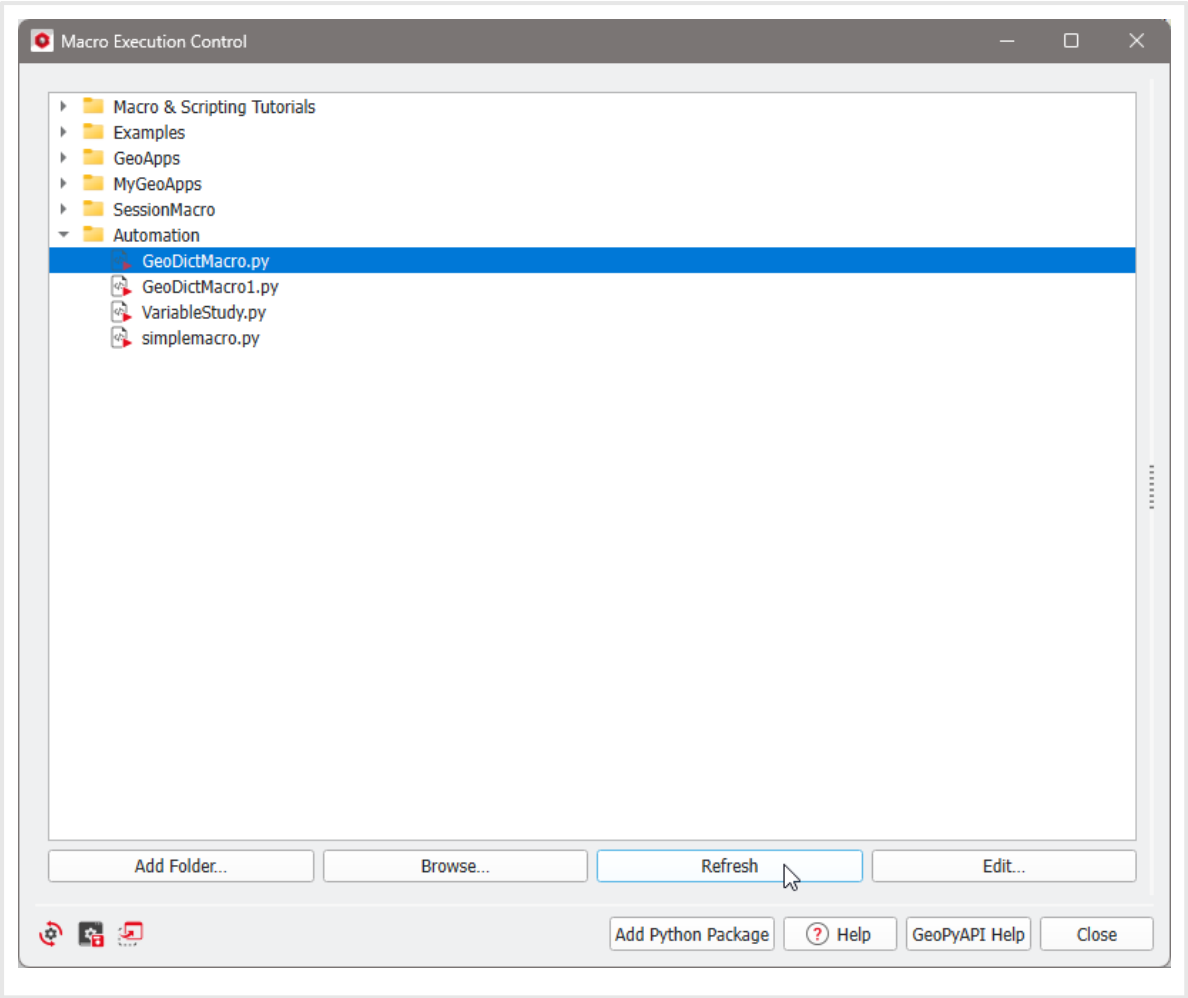


The macro follows the structure: Header={}, Description="' '", Variables={} and the command block.

You can modify directly any parameter or command listed in the command block, or perhaps, introduce a variable.



After modifications, the macro file can be saved with a different name (e.g., GeoDictMacro1.py). Click **Refresh** to have the name of the macro, modified and saved in the project folder, appear in the list of macros in the left panel of the **Macro Execution Control** dialog.



Macro Description

Refer to the descriptions on what the macro will do when executed. There are two kinds of macro descriptions. One is the command preview and only available for simple macros, only containing GeoDict commands. The other is a custom description text that can be changed manually when opening the macro in a text editor.

Command Preview

On the right part of the **Macro Execution Control** dialog, the entries in the upper panel correspond to each one of the gd.runCmd 67 commands, that can be seen when opening the macro with a text editor. They show which command comes next and visualizes for example stepping through the macro as explained here 30.

For **GeoDict:Preferences**, **FiberGeo:Create**, and **GeoDict:LoadGdrFile**, they are as follows:

```
Preferences_args_1 = {
  'Statistics' : {
   'GUIFontSize'                : 9,
   'ColorScheme'                : 'System',    # Possible values: System, Light, Dark
   'SpecifyThreadNumber'        : 'Automatic', # Possible values: Maximum, Manual, Automati
   'RunSolversWithLowPriority' : False,
   'NumberOfThreads'            : 2,
  'JobSystem' : {
   'ReleaseFloatersTimeout'     : 0,
   'IdleTimeout'                : 24,
  'Cluster' : {
  'IO' : {
   'TextEditorFullPath'         : 'C:/Program Files/Notepad++/notepad++.exe',
  'Undo' : {
  'Update' : {
  }
gd.runCmd("GeoDict:Preferences", Preferences_args_1, Header['Release'])
```

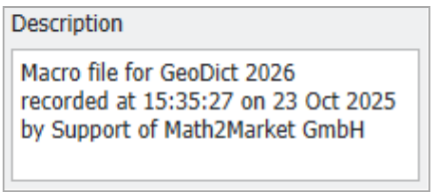
```
   'RandomSeed'                 : 43,
   'IsolationDistance'          : (0, 'm'),
   'MatrixDensity'              : (0, 'g/cm^3'),
  'OverlapMaterial' : {
  'ContactMaterial' : {
   'NumberOfGenerators'         : 2,
  'Generator1' : {
  'Generator2' : {
   'Temperature'                : (293.15, 'K'),
  }
gd.runCmd("FiberGeo:Create", Create_args_1, Header['Release'])
```

```
LoadGdrFile_args_1 = {
  'ResultFileName' : 'FiberGeo.gdr',
  }
gd.runCmd("GeoDictResults:LoadGdrFile", LoadGdrFile_args_1, Header['Release'])
```

Description

The **Description** panel below contains information about the macro. Regarding a recorded macro it gives by default information about when the macro was recorded and who recorded it.

This report content can be found early in the macro, between the triple apostrophes after **Description = ''' '''**, and can be edited at any time after opening the macro with a text editor, by clicking **Edit**.




```
Description = '''
Macro file for GeoDict 2026
recorded at 15:35:27 on 23 Oct 2025
by Support of Math2Market GmbH
'''
```

Fixed and Vary Parameters

For your convenience, the macro block listing the variables (Variables = []) is already created during the recording of a simple macro, but it is initially empty of variables. A simple macro can be transformed into a parameter macro 49. After adding variables, they can be set from the Macro Execution Control 29 using the **Parameters** button.

In the macro, the variable values are described within the brackets of **Variables = []**. They are listed right after the header.

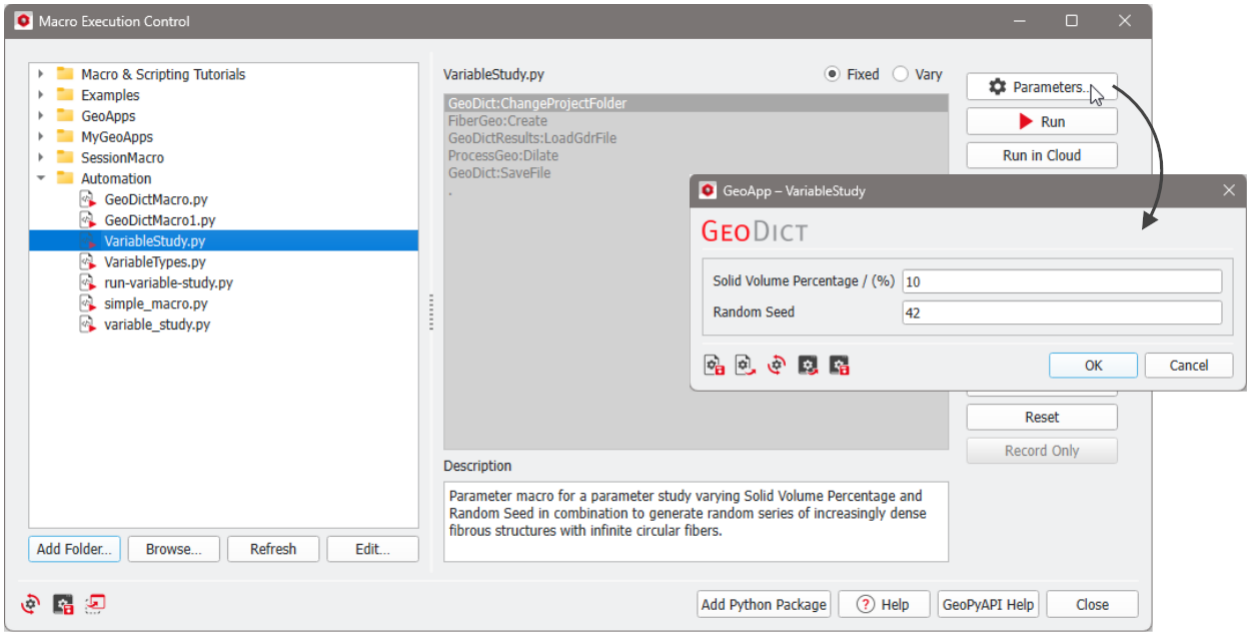
```
Variables = [
   {
     'Name'           : 'gd_SVP',
     'Label'          : 'Solid Volume Percentage',
     'Type'           : 'double',
     'Unit'           : '%',
     'ToolTip'        : 'Solid volume percentage of the created structure.',
     'BuiltinDefault' : 10.0,
     'Check'          : 'min0;max100'
   },
   {
     'Name'           : 'gd_RandomSeed',
     'Label'          : 'Random Seed',
     'Type'           : 'int',
     'Unit'           : '',
     'ToolTip'        : 'Random Seed of the created structure.',
     'BuiltinDefault' : 42,
   },
  ]
```

In the **VariableStudy.py** macro (Download VariableStudy.py), two variables are present as a list of two Python dictionaries. The variables are described by the parameters **'Name'**, and **'Type'** (int : integer) and by the value of the parameter (e.g., **'BuiltinDefault' : 10.0** and **42** here). Click here 58 to learn more about the different variable types .

When a macro contains variables, and thus is a **Parameter Macro**, the **Parameters** button and the **Fixed** and **Vary** radio buttons are available on the right upper side of the **Macro Execution Control** dialog.

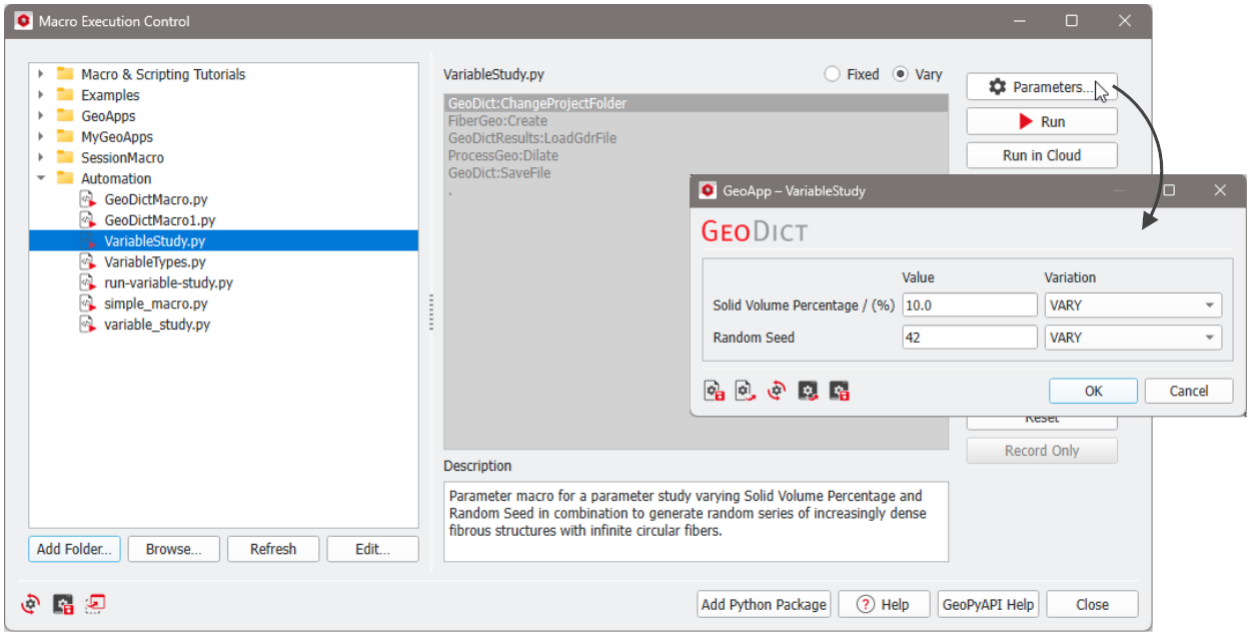
Fixed Parameters

With **Fixed** selected (by default), click **Parameters** to change the parameters for the execution of the macro.



Vary Parameters

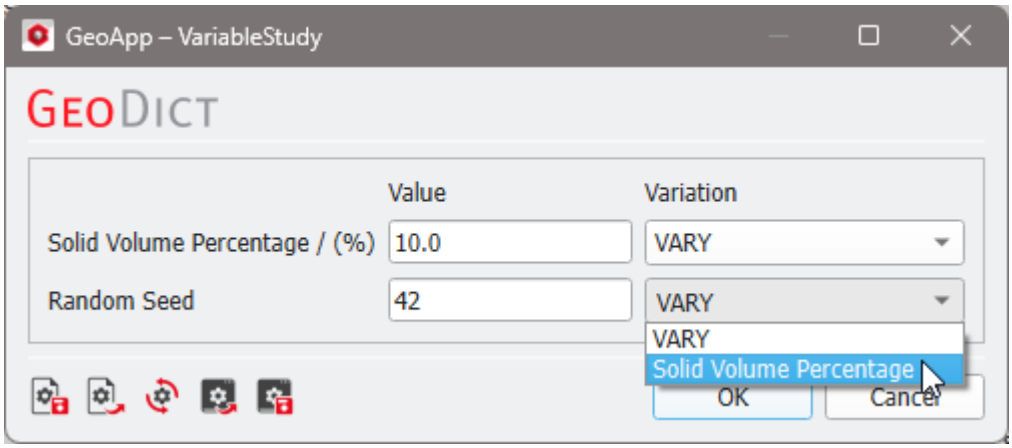
With **Vary** selected, clicking **Parameters** opens a different parameter dialog box where parameter lists can be entered. The macro is executed several times with different parameter values combinations.



Variation - Vary and couple parameters

When editing a parameter macro to run a parameter study in which several variable values should be tried out, the **Value** and the **Variation** for each of the variables must be set. The **Variation** can be set to **VARY** for a list of variable values or can be coupled

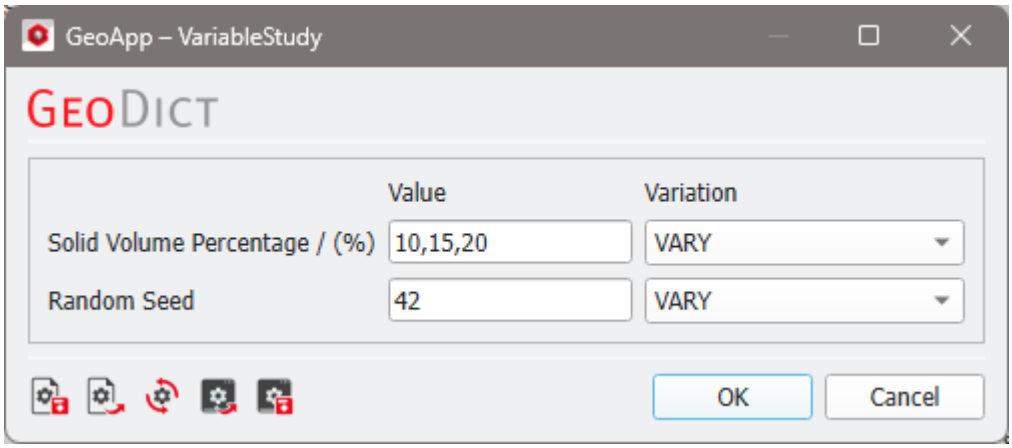
to another variable. Coupled variables are run in a synchronized way. When the value of one variable is varied, the value of the coupled variable is modified accordingly.



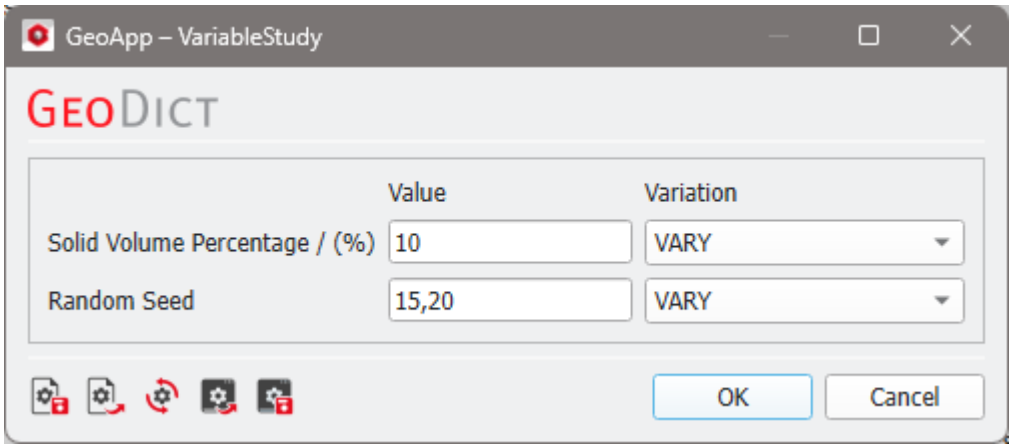
To couple variables, the same number of values must be entered under **Value** in the boxes for every variable.

Observe the effect of choosing **VARY** or coupling to another variable in the pull-down menu for **Variation**:

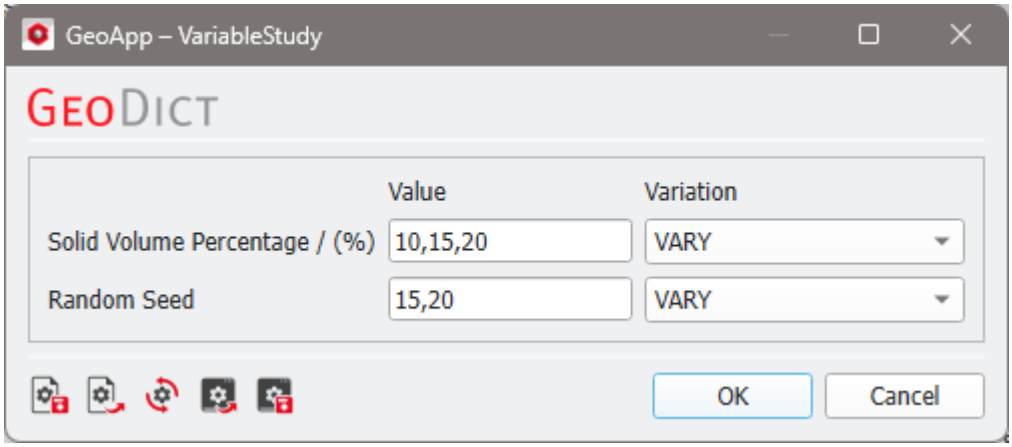
All possible combinations of the **Solid Volume Percentage** values with the single **Random Seed** value are executed, leading to runs with variable values **(10,47)**, **(15,47)**, **(20,47)**. The value of the second variable is kept constant.



Now, all possible combinations of the two **Random Seed** values with the single **Solid Volume Percentage** value are executed, leading to pairs **(10,15)** and **(10,20)**.

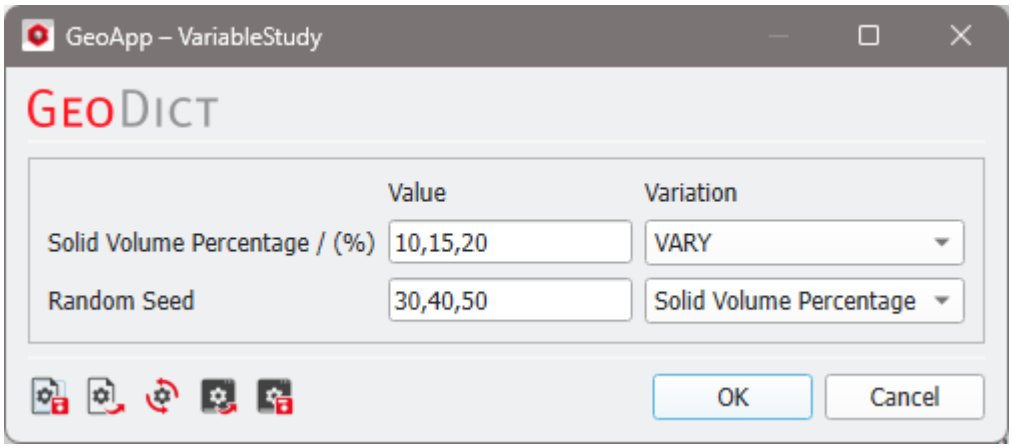


In the following case all possible combinations of the two **Random Seed** values with the three **Solid Volume Percentage** value are executed, leading to pairs **(10,15)** and **(10,20)**, **(15,15)**, **(15,20)**, **(20,15)**, **(20,20)**.

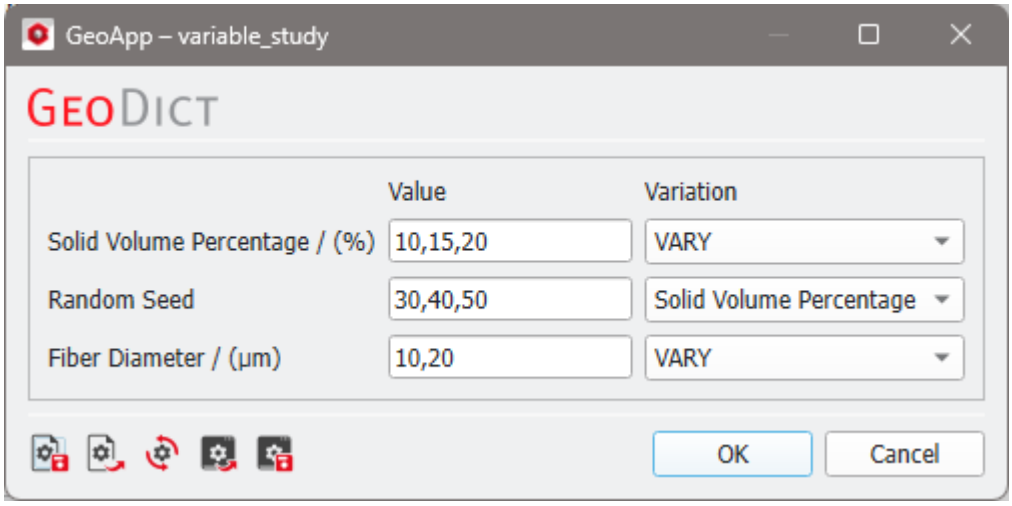


Setting the parameter **Variation** to the other parameter leads to coupled pairs. As mentioned before, the same number of values for every variable must be entered in the boxes.

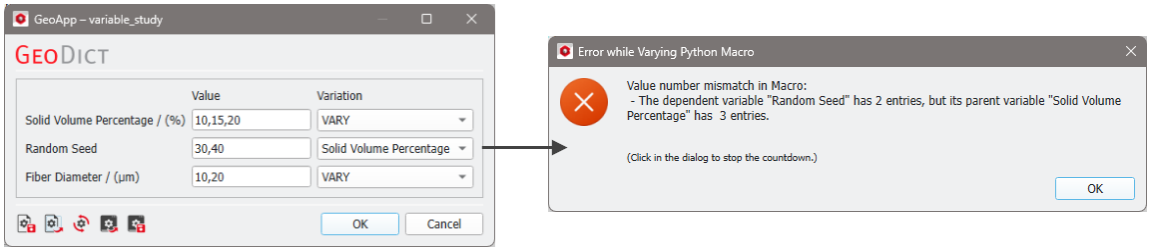
The first values in **Solid Volume Percentage** (**10**) and **Random Seed** (**30**) are coupled with each other, as well as the second values with each other (**15** and **40**), and the third values with each other (**20** and **50**), resulting in the combinations **(10,30)**, **(15,40)**, and **(20,50)**.



If a parameter macro contains more than two variables, not all variables must be coupled. Coupling **Random Seed** to **Solid Volume Percentage** and leaving **Fiber Diameter** to **VARY**, leads to the combinations **(10,30,10)**, **(10,30,20)**, **(15,40,10)**, **(15,40,20)**, **(20,50,10)** and **(20,50,20)**.



An error message appears after clicking **Run** in the **Execute Parameter Macro** section, when the values entered in the parameters dialog box are invalid.

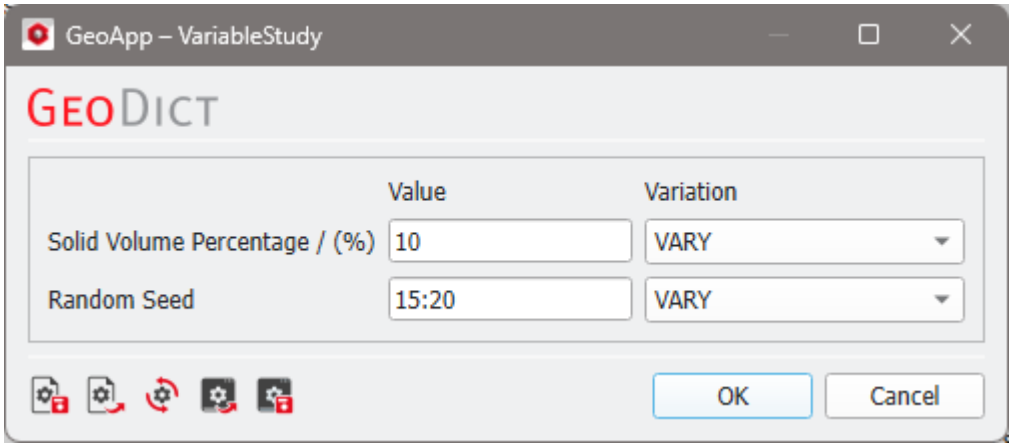


Otherwise, clicking **Run** starts the execution of the parameter macro.

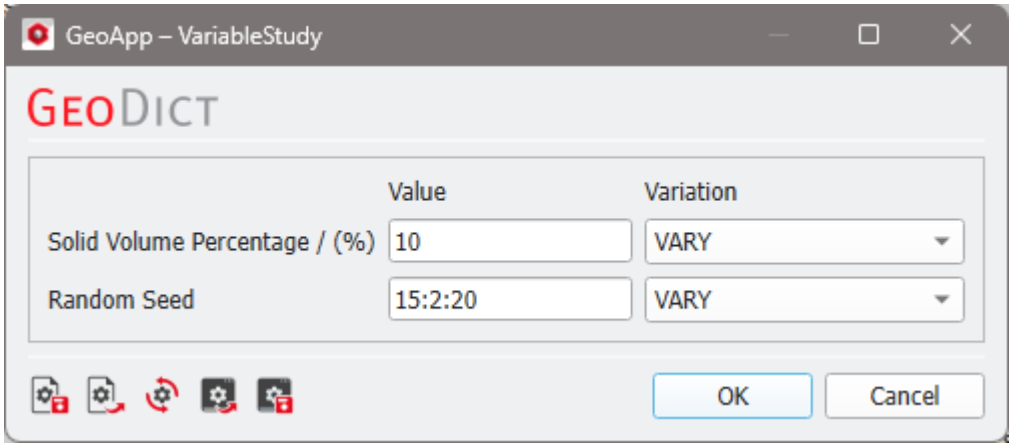
Enter lists using Start:Step:End Syntax

It is also possible to enter a range of parameter values for **Value** using the notation **start:step:end**. This is useful if longer lists of variable values must be entered.

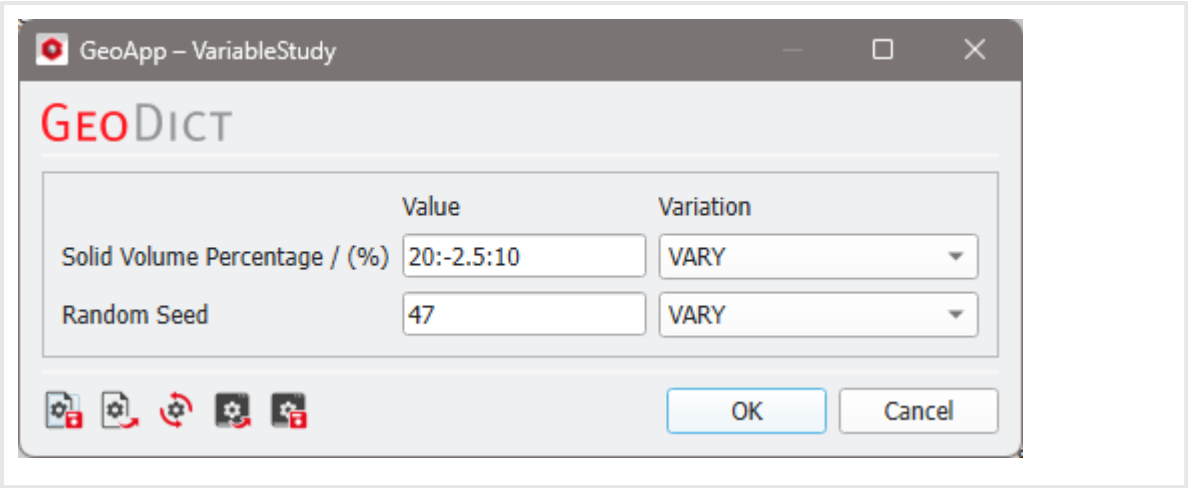
The notation **15:20** means that all the values between 15 and 20 are taken for the computation. This results in the combinations **(10,15)**, **(10,16)**, **(10,17)**, **(10,18)**, **(10,19)**, and **(10,20)**.



Also, the stepping can be set using the colon notation. The notation **15:2:20**, meaning to start from 15, and to take only every second value until 20 is reached, results in the combinations **(10,15)**, **(10,17)**, and **(10,19)**.

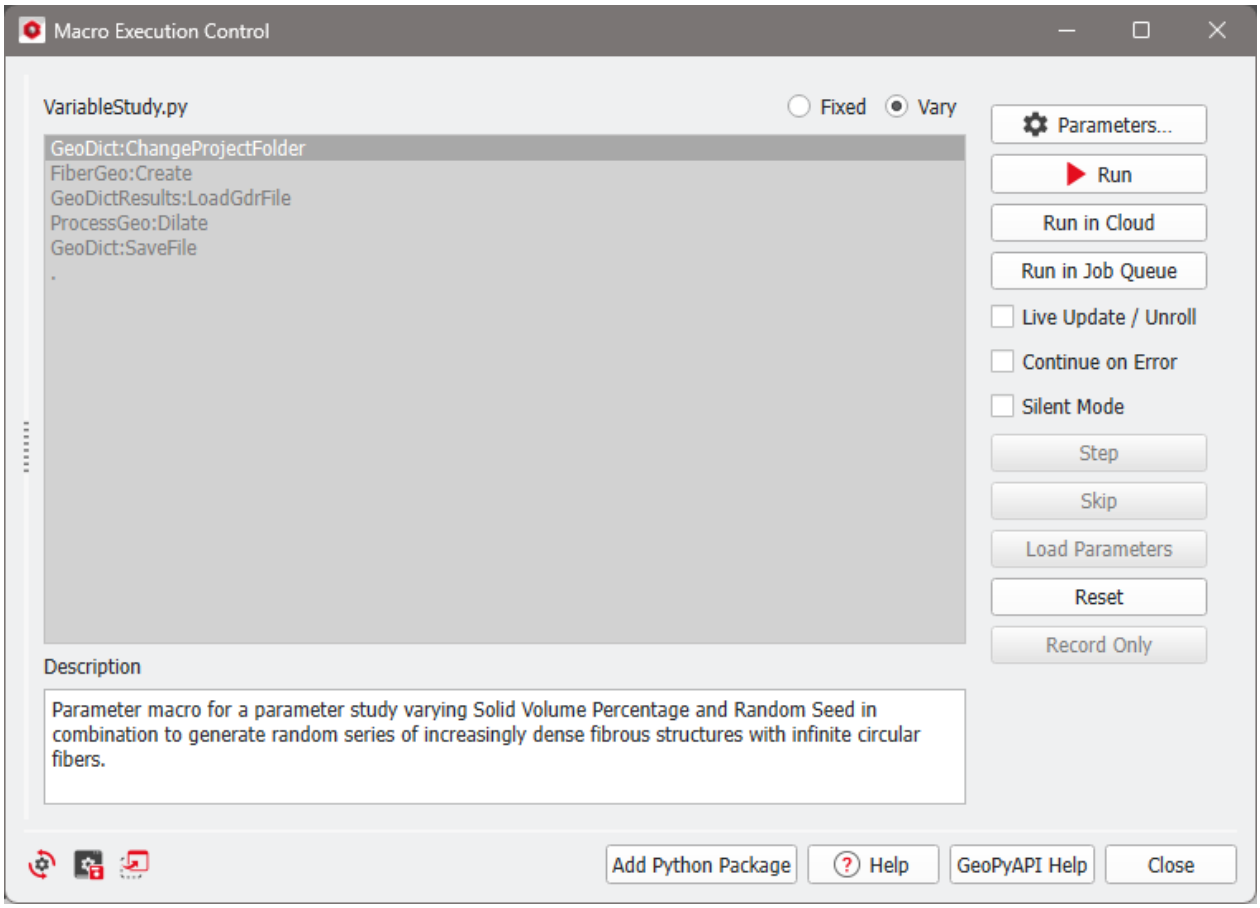


For the stepping value, negative values can also be used, if the start value is bigger than the end value. If the variable is a floating number, a floating point can be used as stepping value. **20:-2.5:10**, meaning to start from 20, and to take only every 2.5^{th} value until 10 is reached, results in the combinations **(20.0,47)**, **(17.5,47)**, **(15.0,47)**, **(12.5,47)**, and **(10.0,47)**.



Run Macros

There are different possibilities to run a macro. You can run it locally, in the cloud or on a job server. You can run it completely, only step-wise or simply record the macro execution in another macro without executing it.



Run complete macro

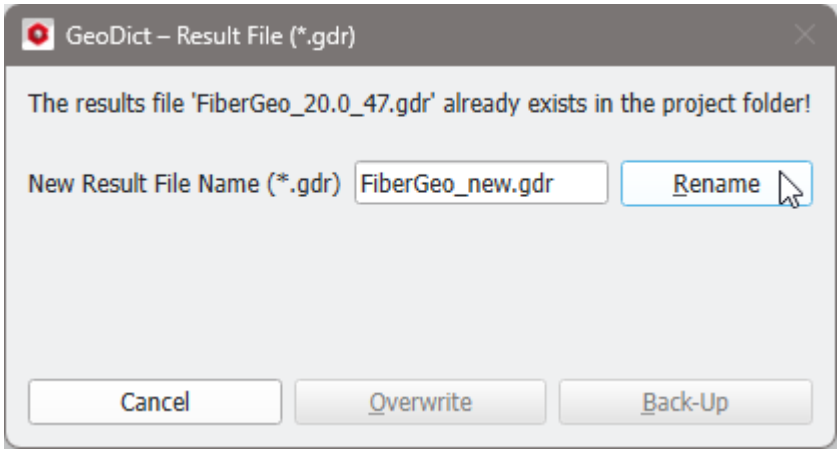
To execute the complete macro on the current machine, click **Run**.

Click **Run in Cloud** to run the simulation in the GeoDict cloud, see the **Cloud** chapter for details. To run the macro on machines connected in your local network, click **Run in Job Queue**, also explained in more detail in the **Job Queue** chapter. If interested in cloud simulations or job queuing contact Math2Market to apply for a GeoDict cloud or job queuing license.

With **Live Update/ Unroll** checked, every step is shown in the GUI. Additionally, all commands executed in the macro are recorded to the Session Macro 35, instead of only recording the GeoDictMacro:Execute command. However, the execution of the macro is faster if this box stays unchecked.

The **Continue on Error** checkbox below can only be checked if **Vary** is selected. Check **Continue on Error** to execute all parameter combinations entered to the **Parameter** dialog box that work and not only all up to the parameter that results in an error. For example, if the parameters 10, -5, 20 are chosen for the Object Solid Volume Percentage, the macro executes only for SVP=10. When **Continue on Error** is checked, it is also executed for SVP=20.

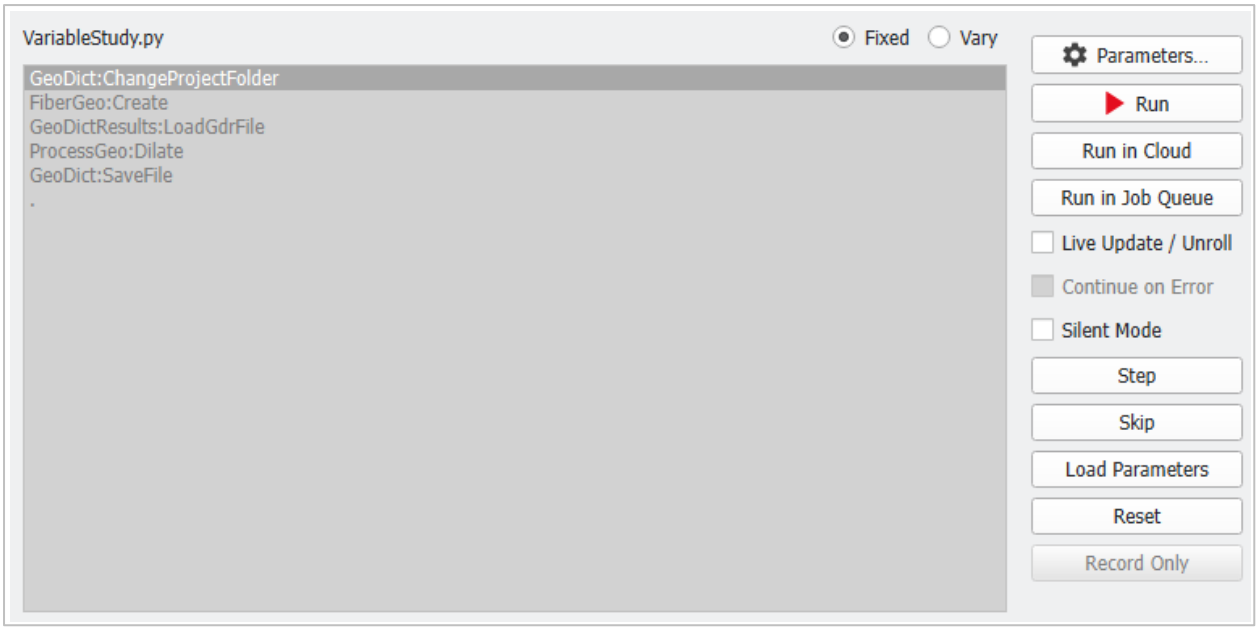
When the executed macro includes a command for which you must intervene (such as the saving of a result file when one with the same name already exists), a message appears to decide whether the data should be rewritten or should receive a new name. A lack of reaction within 20 seconds results in the existing data being automatically saved with a suffix (current time) in a new folder called **00GeoDictBackUp**. The message waiting time can be changed in the settings dialog to be found by selecting **Settings** → **Settings...** from the menu bar.



If running the macro in **Silent Mode** no message boxes are shown. If only parts of the macro should not show messages, you can use the gd.getBlocker 95 API command within the macro, instead.

Running the macro step-wise

Alternatively, the macro's key commands can be executed step-by-step when clicking **Step** instead of **Run** (only available if **Fixed** 23 is selected).



While stepping through the macro, the GeoDict's GUI main screen remains active, so that it is possible to see and save intermediate results, as well as change the rendering from 2D to 3D.

The execution of the macro can be further controlled with **Skip**, **Load Parameters**, and **Reset**. During a step-by-step execution, the highlighted key command in the description area is jumped over when clicking **Skip**.

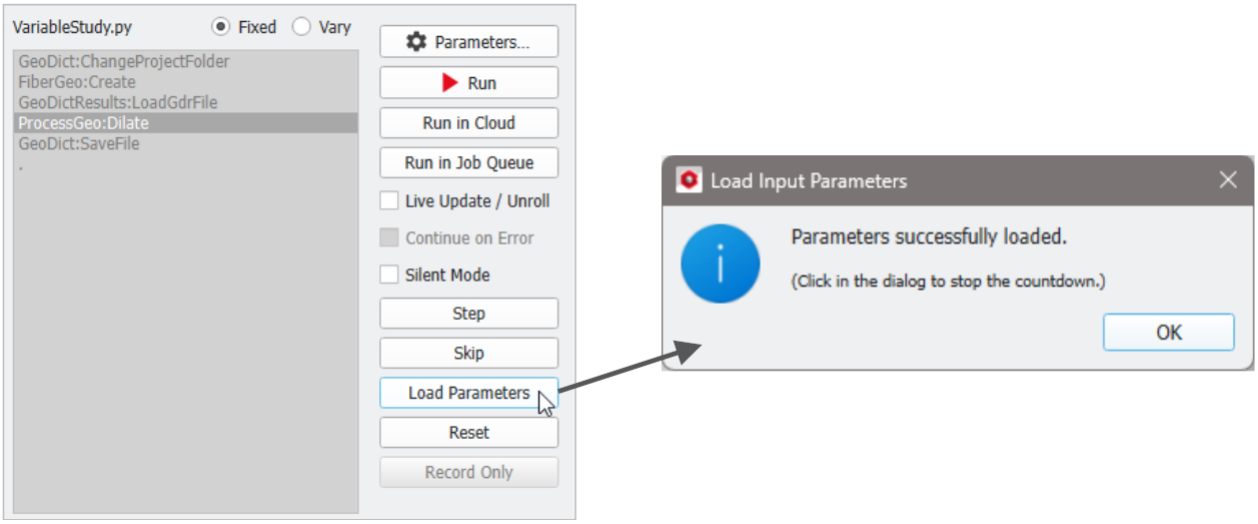
You must consider the consequences that the skipping of a command has. For example, an error message appears when skipping the creation of a new project folder for the data, so that the data is actually saved in the current project folder and then, trying to leave the (not created and not existing) project folder, and move up the folder path.

Clicking **Load Parameters**, the parameters from the highlighted macro command are entered for inspection in the corresponding parameters dialog box or in the module section.

However, when later executing the extracted macro command, the parameters continue to be taken from the saved macro. Modifying parameters in the inspected dialog box has no effect on the previously recorded macro or in the ongoing execution of the macro.

For example, when clicking to load the parameters from the command **ProcessGeo Dilate** the parameters used for Dilate MaterialID, Coating MaterialID, and Dilate by..., during the recording of the macro, are directly entered in the ProcessGeo section.

Loading the parameters might be interesting if you decide to abandon the execution of the macro at a given command, and to post-process the structure by modifying its parameters directly in the module's GUI, to obtain a different result.



When clicking **Reset**, the first key command in the description area is highlighted again so that the macro can be executed stepwise from the beginning.

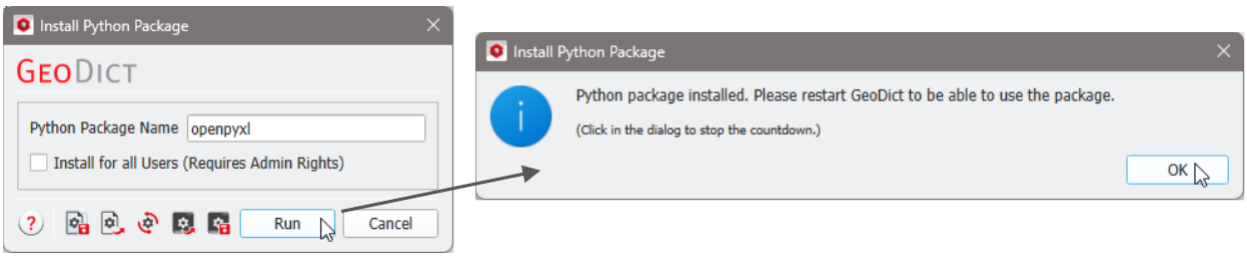
Recording macro execution

Click **Record Only** while recording a macro to record the commands and the variables edited in the **Parameters** dialog of the selected macro in the **Macro Execution Control**. Find an example in Vary Parameters from GeoPy 56

Adding Python Packages

To install additional Python packages click **Add Python Package** in the **Macro Execution Control** dialog.

Fill in the name of the desired Python package. Clicking **Run** installs the package automatically. Owning admin rights, it can be **install**ed **for all Users.** If installed for all users, the package is installed in the GeoDict installation folder (e.g., C:\Program Files\Math2Market GmbH\GeoDict 2026\Python\lib\site-packages). If installed only on the local machine, it is installed to the Python folder inside the GeoDict settings folder (C:\Users\Username\GeoDict2026\Python). After installation, restart GeoDict to use the new package.



It is also possible, to install Python packages offline, if downloaded before. For this, run a Python macro 29. The macro must contain the following code:

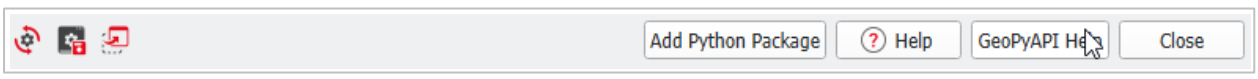
```
InstallPyPackage_args = {                # define parameters dictionary

  'Name' : 'dummy.whl',                  # instead of dummy.whl enter
                                         # the file path of the whl
                                         # file to install

  'Global' : False,                      # Global is the key for the
                                         # checkbox "Install for all
                                         # Users". False means, the box
                                         # is not checked. If changed
                                         # to True, Admin Rights are
                                         # required to install for
                                         # all users.

  'Mode' : 'LocalInstall'}               # Select the mode LocalInstall
                                         # to install the package offline

gd.runCmd("GeoDict:InstallPyPackage",    # execute the installation
InstallPyPackage_args)
```

The Python dictionary containing these keys can also be obtained by installing a Python package using the button **Add Python Package** described above, while a macro is recorded as described here[11]. Then, the value for **Mode** is **'Install'**. The third mode, that can be selected is **'Download'**. If a Python package should only be downloaded and not installed, use the installing Python package dictionary as follows:

```
InstallPyPackage_args = {                # define parameters dictionary

  'Name'    : 'openpyxl',                # instead of dummy enter the
                                         # name of the Python package
                                         # to download

  'Global'   : False,

  'Mode'    : 'Download'}                # Select the mode Download
                                         # to only download the package

gd.runCmd("GeoDict:InstallPyPackage",    # execute the download
InstallPyPackage_args)
```

GeoPyAPI Help

Click **GeoPyAPI Help** to open an overview about all GeoDict Python API[66] commands. This overview is opened as an *.html file in the default browser. Note, that this is no internet link but instead an *.html file shipped with GeoDict.



GeoPy API Documentation for GeoDict2026

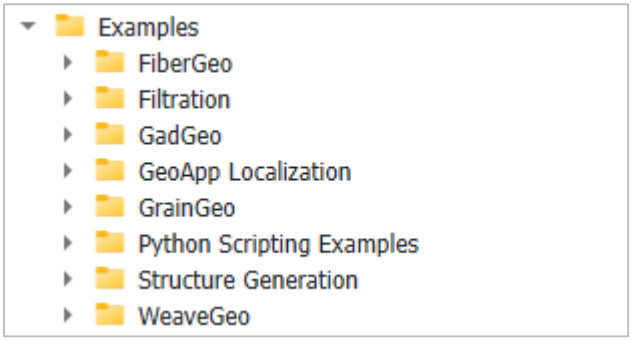
Class GD:

The class that allows interfacing with GeoDict. An instance is automatically created and accessible under the name gd in a GeoPy Macro.

Function Name	Arguments	Return Value	Description
get2DViewAsPlot	dir : int slice : int topToBottom : bool	dict	Returns the plot dictionary to put in a gdr

1.3.3 Open Scripting Examples

Clicking **Open Scripting Examples** in the macro menu or directly unfolding **Examples** in the Macro Execution Control gives access to several GeoPy scripts shipped with GeoDict. These scripts are more advanced than the ones to found in the **Macro & Scripting Tutorials** 34 folder. These Python scripts also use other GeoDict modules.

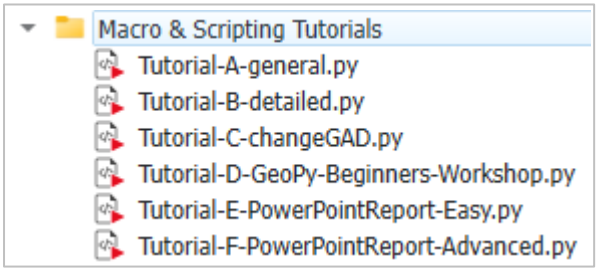


When selecting one of the available macros, the description 20 area displays a report about the macro. In the macro, this report content can be found between the triple apostrophes after Description = ''' ''', and can be edited at any time after opening the macro with a text editor.

Click **Edit** to inspect the macro content in a text editor.

1.3.4 Open Scripting Tutorials

Clicking **Open Scripting Tutorials** in the macro menu or unfolding **Macro & Scripting Tutorials** in the Macro Execution Control shows a list of preinstalled macro tutorials.



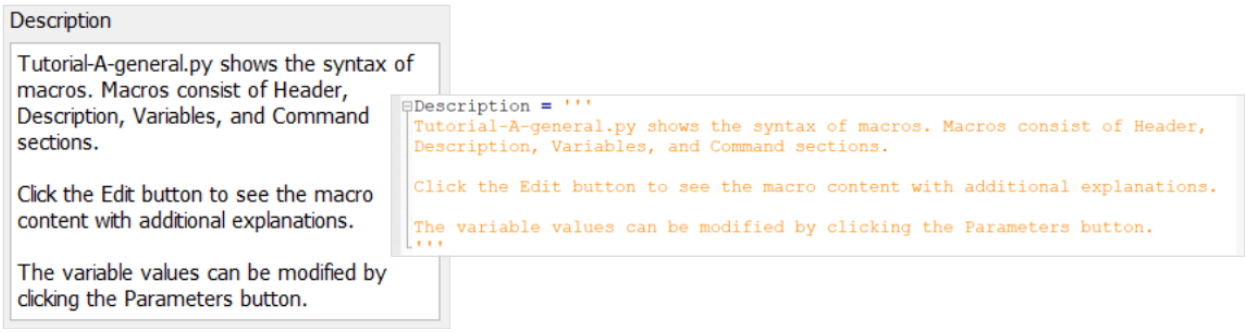
The tutorial macros A, B, C and E need only a GeoDict Base license for execution. The tutorials D and F also need the modules FiberGeo and FlowDict and are the macros created in the workshop videos available on the Math2Market YouTube channel. Find the corresponding links

in the macros by opening them in a text editor clicking Edit. There, you can also find detailed explanations for all steps.

More advanced example macros can be found in the subfolder Python Scripting Examples. These Python scripts also use other GeoDict modules.

All tutorials have detailed descriptions and thus can be very helpful for getting started with editing Python macros. When selecting one of the available macros, the description area displays a report about the macro. In the macro, this report content can be found between the triple apostrophes after Description = ''' ''', and can be edited at any time after opening the macro with a text editor.

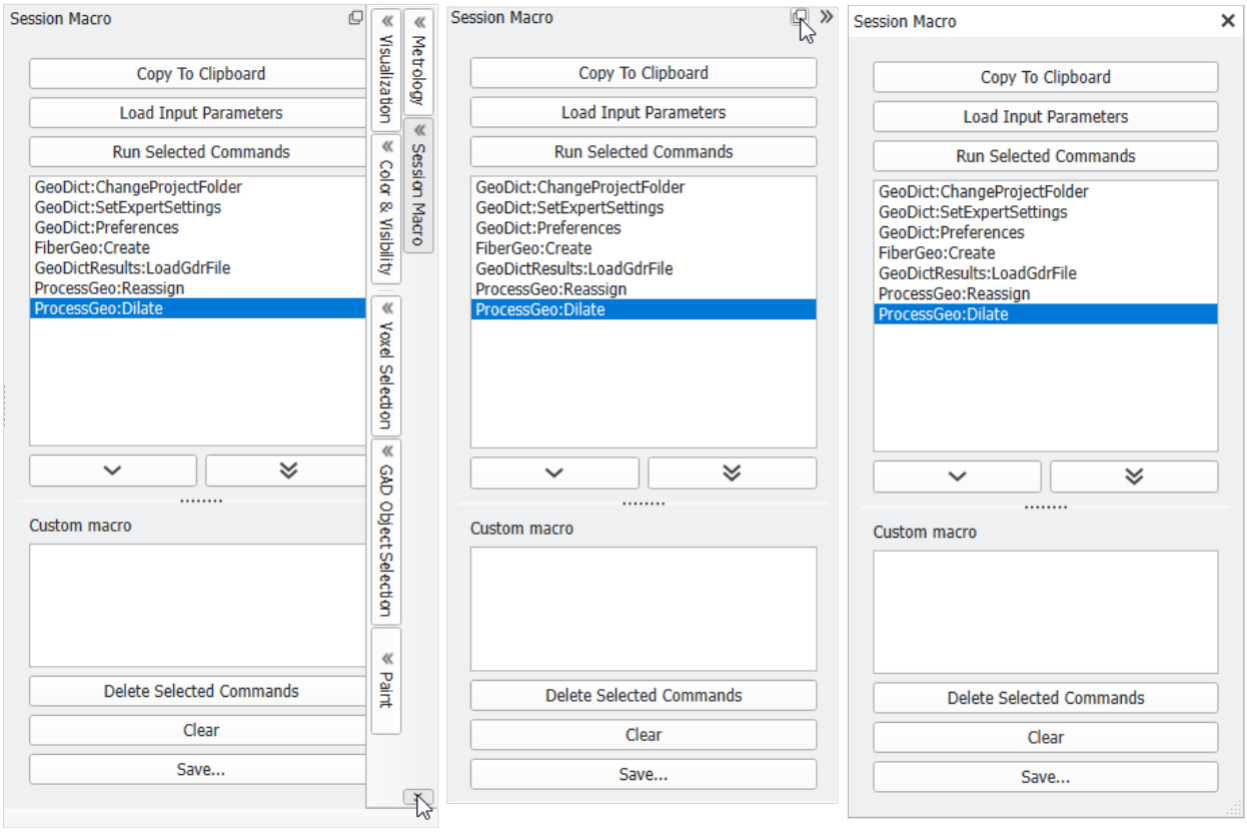
For the Tutorial-A-general.py macro, the text in the macro and in the description area are shown here.



1.3.5 Session Macro

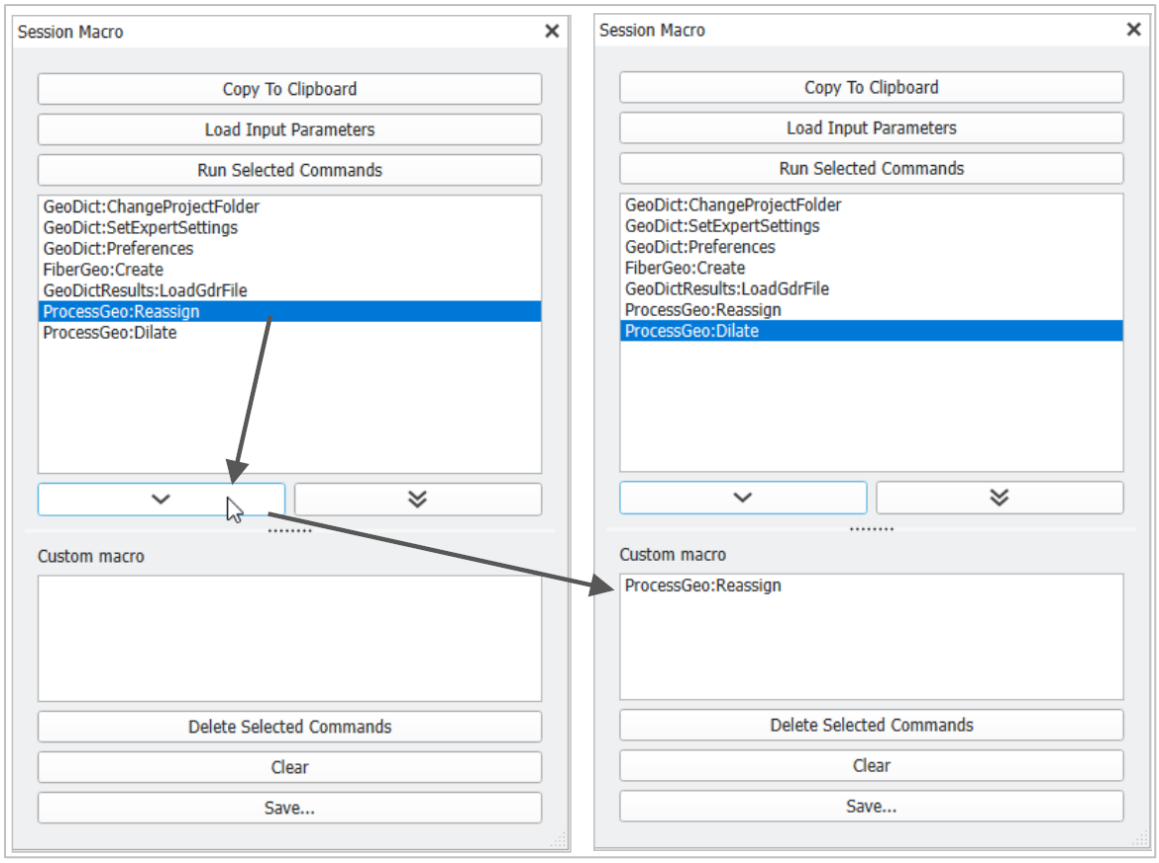
From the moment in which you begin a session with GeoDict, all commands used are internally recorded and stored in the **Session Macro**. You may decide to select some of these recorded commands, create a macro that combines them, and save this macro for later use.

After selecting **Macro → Session Macro...** in the menu bar, the **Session Macro** side bar opens. It can also be opened by clicking on the side bar tab on the right of GeoDict. If the GeoDict window is not maximized, the Session Macro tab may not be shown. It can be found by clicking at the bottom of the side bar. You can also undock the **Session Macro** panel and turn it into the **Session Macro** dialog by clicking in the upper right corner. Although it is still minimized if the GeoDict GUI is minimized, the dialog can be moved independently on the screen.



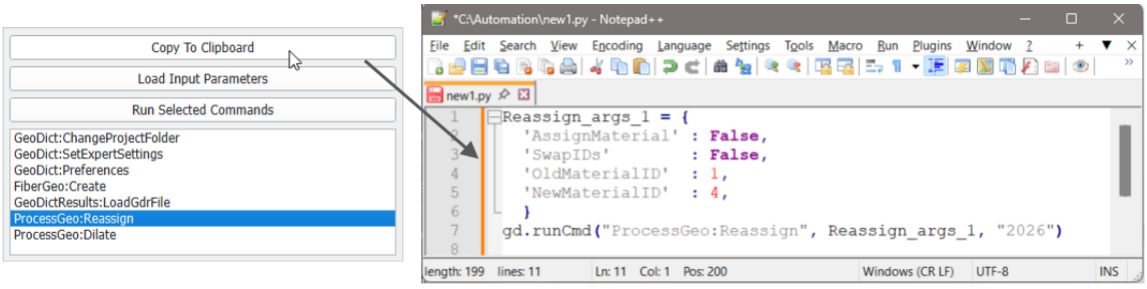
Select the commands to record

The commands used during the session appear in the upper panel of the **Session Macro** dialog and can be selected (highlighted). Click the single arrow  to move the selected commands to the lower panel (**Custom macro**) in the desired order. To move several commands at once, press and hold Ctrl while selecting the desired commands.



To choose and move all commands from the upper panel at once, click the double arrows  instead.

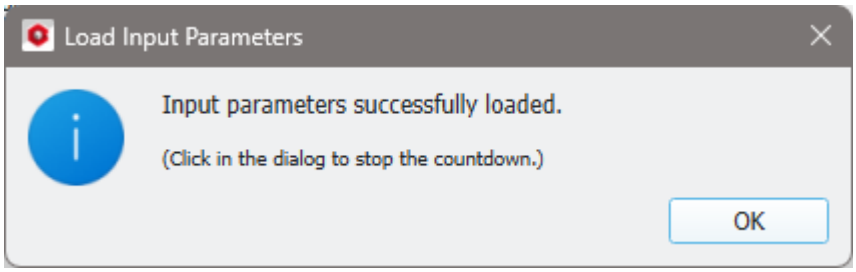
Clicking **Copy To Clipboard** copies the highlighted commands from the upper panel to the clipboard. From there, you can paste them to an editor.



The commands can be removed from the lower panel by highlighting them and clicking **Delete Selected Commands**. To remove all commands at once, click **Clear**.

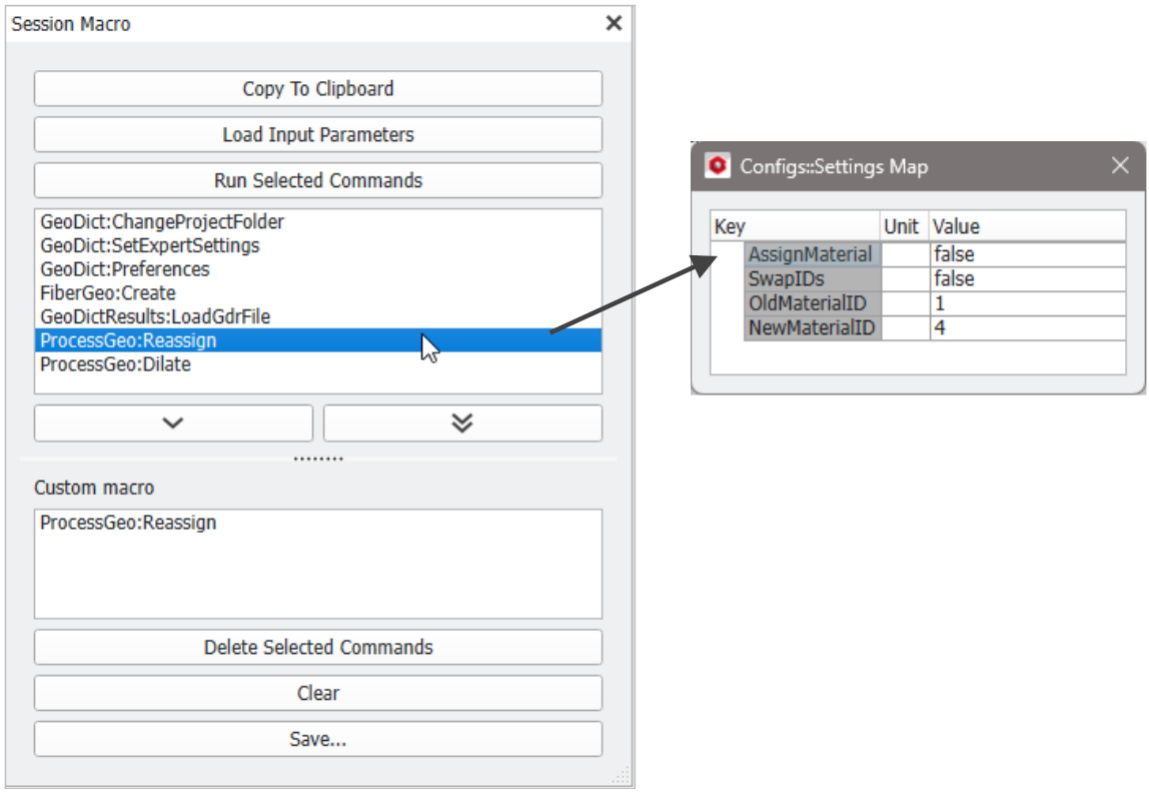
Load input parameters and directly run a command

Click **Load Input Parameters** to only load the parameters of a single highlighted command in the corresponding parameters dialog box in the module section.



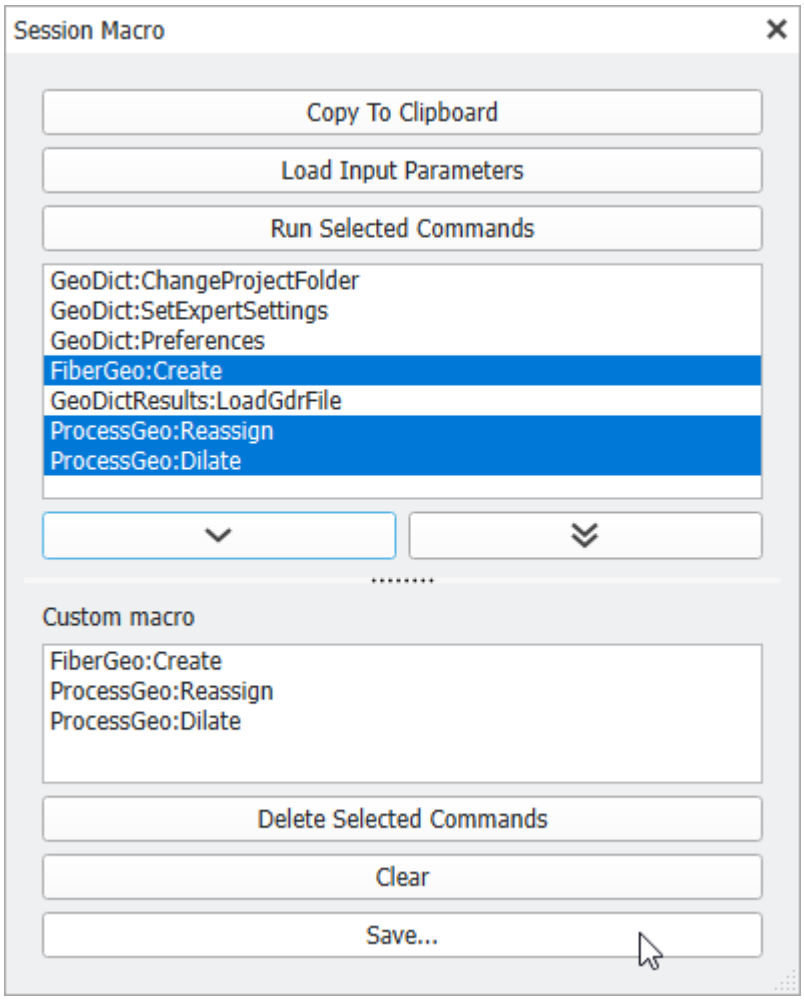
To run commands again without saving them to a macro, highlight the desired commands in the upper panel and click **Run Selected Commands**.

Double clicking on a command, whether in the upper or in the lower panel, shows the corresponding settings map in a new dialog.

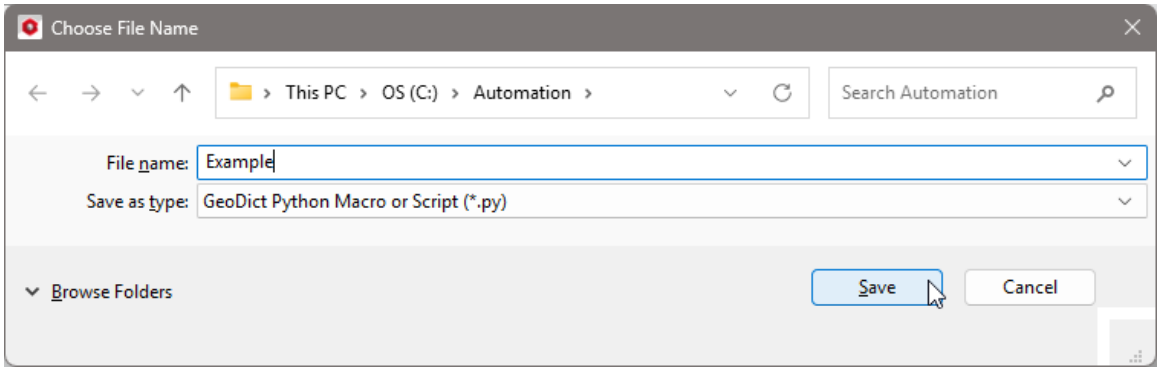


Save your custom macro

After selecting and adding the commands click **Save**.



In the appearing dialog, choose a file name and the desired folder where the macro will be stored.



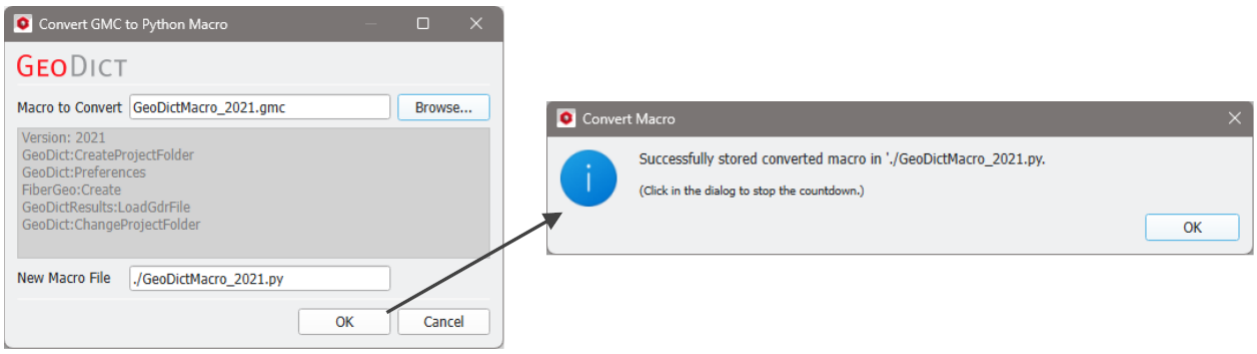
1.3.6 Convert GMC to Python Macro

GeoDict also ships with a compiler that can convert your old GMC macros to Python macros.

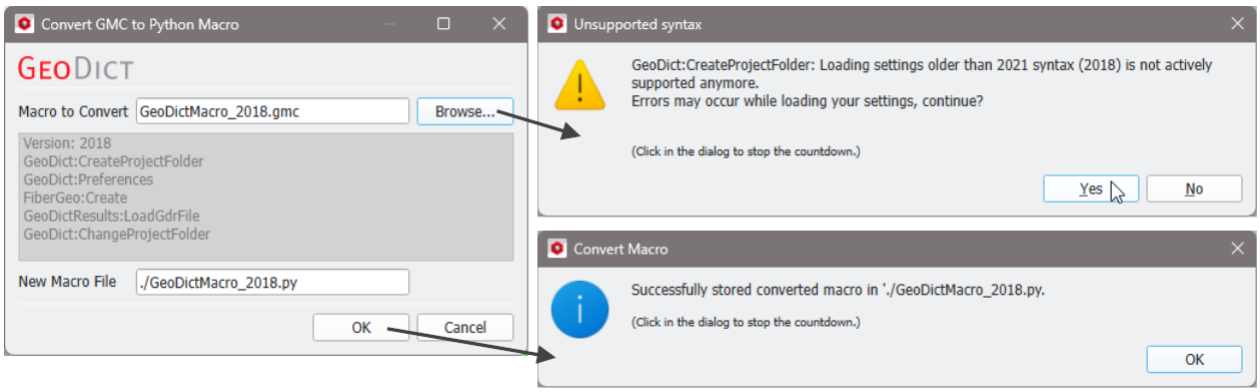
Important! GeoDict 2026 does not effectively support macro syntax older than GeoDict 2021. Errors may occur, when running macros with older syntax or loading

parameter settings from them.

Select **Macro → Convert GMC to Python Macro** in the menu bar. Click **Browse...** in the dialog box to select the *gmc macro to be converted and click **OK**. The new Python macro can be found in the same folder as the GMC macro.



If the GMC macro has syntax older than GeoDict 2021, a warning message appears when selecting it. You can still click **OK** and try if the macro runs through. If this is not the case, you can either record the commands again or if you have a license for an older GeoDict version (GeoDict 2020 or GeoDict 2021) you can convert the GMC macro to GeoDict 2021 syntax and then convert it to Python.

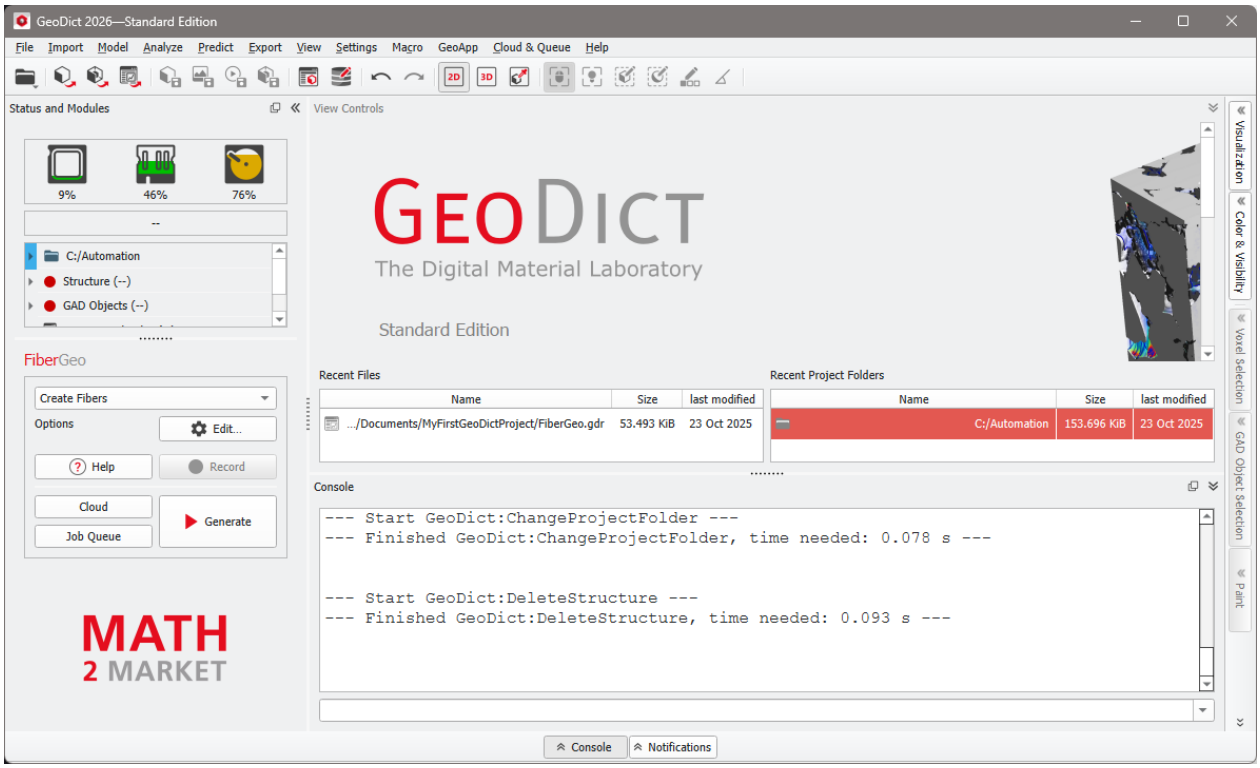


1.3.7 Re-execute Last Python Script

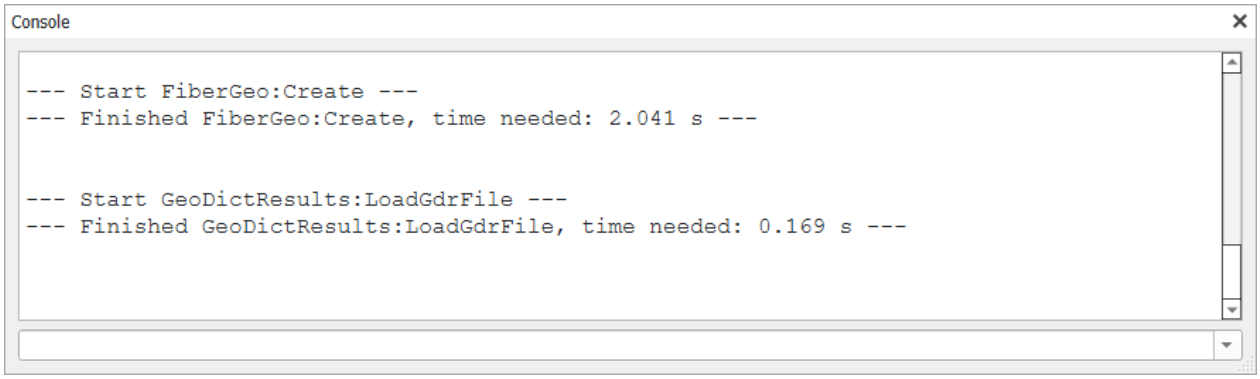
To quickly execute again the last Python script, select **Macro → Re-execute Last Python Script** or press **F9**. The python script is simply executed again without other selections. This is very helpful, if you are debugging your Python script.

1.4 Console and Notifications

The console and notifications panels are located at the bottom of the GUI, under the visualization area. The **Console** panel is a textual GUI window and log viewer. The console allows you to read the system logs, help find certain ones, monitor them, and filter their contents. Additionally, you can enter and execute python commands directly in the console. For examples, see typing commands in console below 44.



The **Console** and **Notifications** panels can be undocked and turned into a dialog that can move around. Undocking is done by clicking the  icon, located at the top right after expanding. Although still minimized if the GeoDict GUI is minimized, the dialog can be moved independently on the screen.



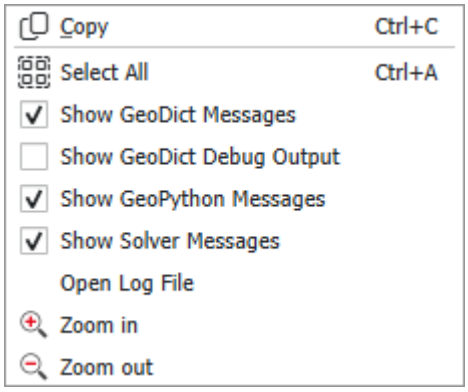
To connect the dialog with the GUI again, drag and drop it to its place at the bottom of the GUI or simply close the dialog, by clicking on  .

Console

GeoDict provides an interactive console within the GUI. All commands running from the GUI are displayed in the console.

The console panel can be folded and unfolded by clicking on the Console icon in the bottom of GeoDict.

A right click in the console panel provides further options to filter and read the information displayed. Text in the console can be marked with the cursor and exported to another program by using **Copy**. With **Select All** the whole text in the console is marked.

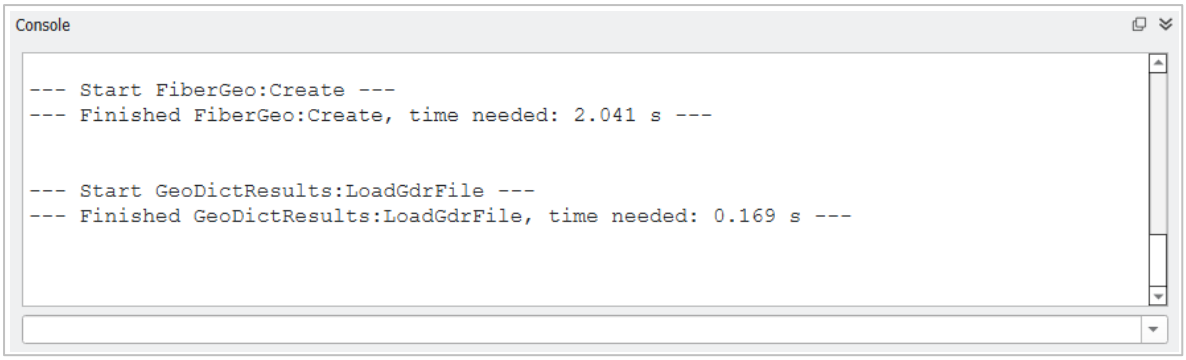


The console content is grouped in four different kinds of messages. Each group has its own color and you can decide which of them to show by checking the corresponding boxes.

- **Show GeoDict Messages:** all direct GeoDict messages. These messages are black for light mode and white for dark mode.
- **Show GeoDict Debug Output**: GeoDict commands may have additional output intended to help localizing bugs and measuring performance. These messages are blue.
- **Show GeoPython Messages**: show output from GeoPy macros and functions. These messages are green.
- **Show Solver Messages**: show messages from GeoDict solvers, e.g., convergence progress. These messages are purple.

During each GeoDict session a log file, that contains all outputs from the console, is written and saved in %username%/GeoDict2026/log (Windows) or %username%/.geodict2026/log (Linux). With **Open Log File** it is opened in the chosen text editor 47.

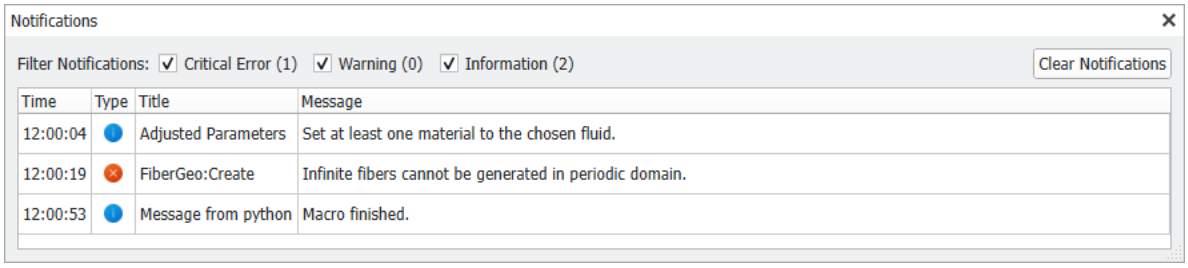
Use **Zoom in** and **Zoom out** to enlarge or reduce the font size of the console.



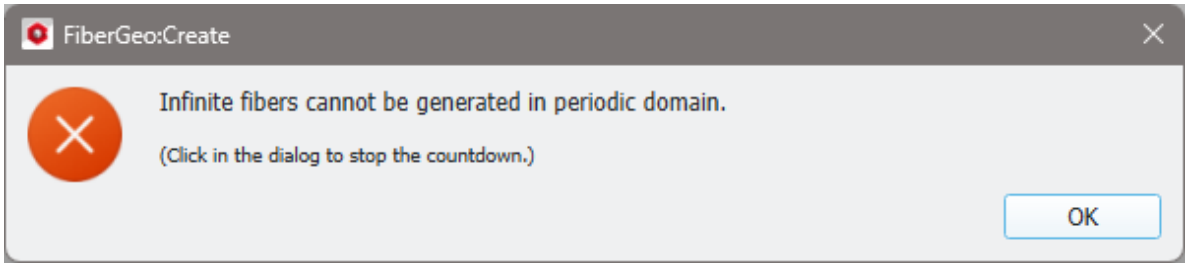
Notifications

The **Notifications** panel can be opened and closed in the same way by clicking on the Notifications icon.

Messages from GeoDict, which are shown in the console, are also shown in the **Notifications** panel.

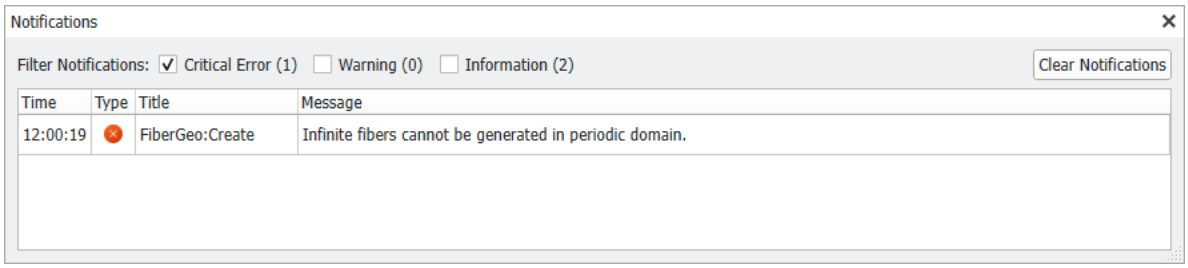


The list shows the **Time**, when a message was raised, the icon shows the **Type** of the message, also the **Title** and the **Message** itself are given. The message is fully displayed in the tooltip, which opens when resting with the cursor over the message. Double-click on a message to open the message dialog again. This is helpful, when a message box was closed, and you want to re-read the message. Another advantage is, that you do not have to scroll through the console, where other output from GeoDict is listed.



With a right click in the corresponding field in the **Notifications** panel the title and the message text can be copied to the clipboard to save them, e.g., in a text editor.

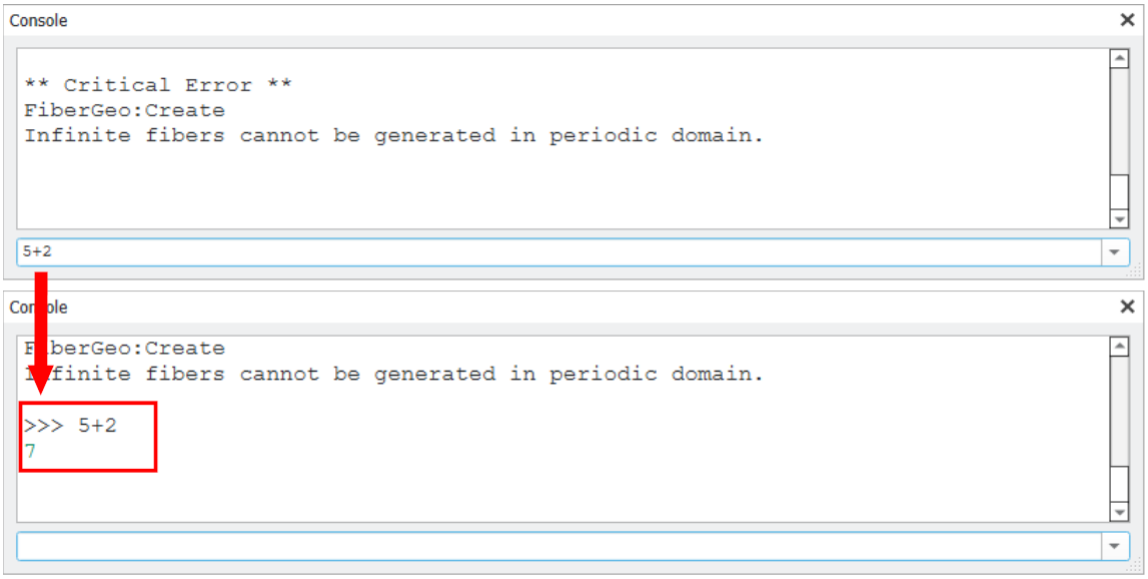
With the filter options it is possible to show only **Critical Errors**, **Warnings** and/or **Information**.



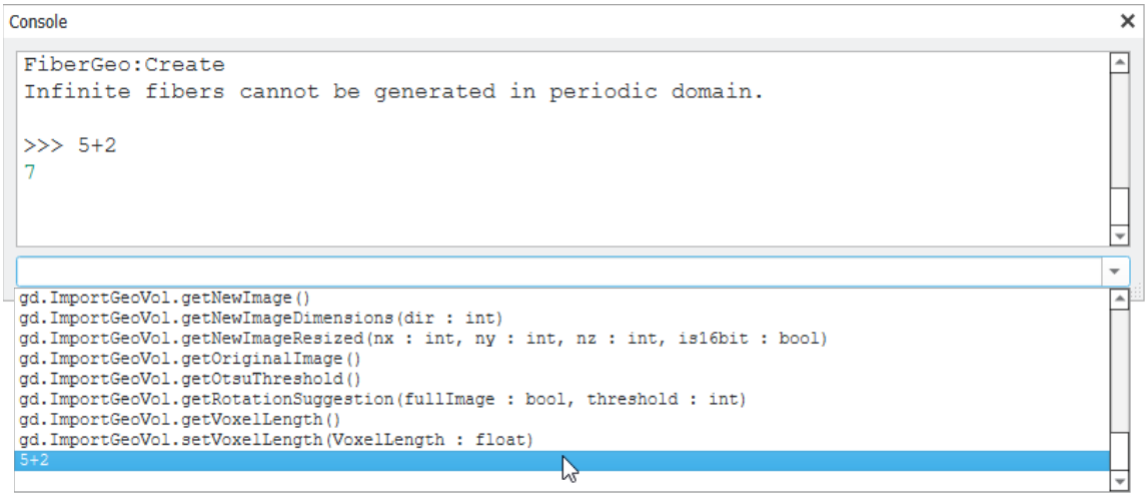
Click on **Clear Notifications** to delete all messages from the list.

Typing Python commands in the console

The box below the console can be used to run Python commands. One command line at a time can be inserted, and it is run by pressing **Enter** on the keyboard.



Unfolding the pull-down menu of the box shows the last used commands and some standard commands from the GeoDict Python API 66.



Besides, variables can be used. Store information in a variable for later use as, for

example, the Python dictionary of the current FiberGeo parameters:

```
FiberGeo_Create = gd.getCurrentSettings('FiberGeo:Create')
```

Typing the variable name again displays the value in the console. In the example, the Python dictionary from the FiberGeo **Create Options** dialog is shown.

```
FiberGeo_Create
```

```
Built-in default settings restored.

>>> FiberGeo_Create = gd.getCurrentSettings('FiberGeo:Create')
>>> FiberGeo_Create
{'ResultFileName': 'FiberGeo.gdr', 'MaterialMode': 'Material', 'MaterialIDMode':
```

The variable value can be changed at any time by assigning a new value to the variable, using the equal sign. Changing only one entry of a dictionary is done by referring to the entry's key in square brackets. The new value is assigned using the equal sign.

```
'StoppingCriterion': 'SolidVolumePercentage', 'SolidVolumePercentage': (10, '%'),
```

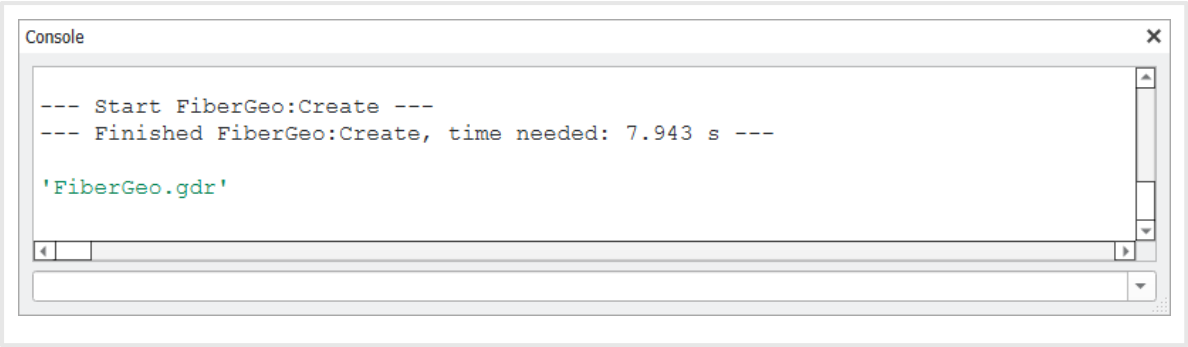
```
FiberGeo_Create['SolidVolumePercentage']= (50,'%')
```

```
FiberGeo_Create
```

```
'StoppingCriterion': 'SolidVolumePercentage', 'SolidVolumePercentage': (10, '%'),

'StoppingCriterion': 'SolidVolumePercentage', 'SolidVolumePercentage': (50, '%'),
```

Now FiberGeo can be run with a solid volume percentage of 50 instead of 10, using the Python API command **gd.runCmd()** which is described in the API for GeoDict commands 67.

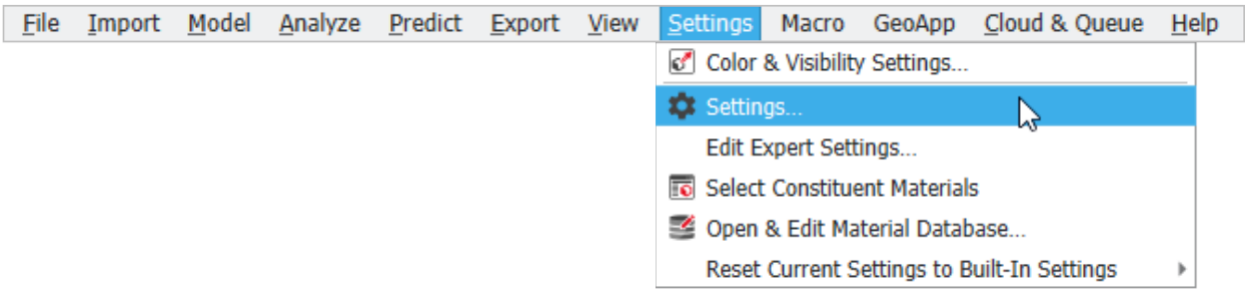
```
gd.runCmd('FiberGeo:Create', FiberGeo_Create, '2026')
```


Console
--- Start FiberGeo:Create ---
--- Finished FiberGeo:Create, time needed: 7.943 s ---
'FiberGeo.gdr'

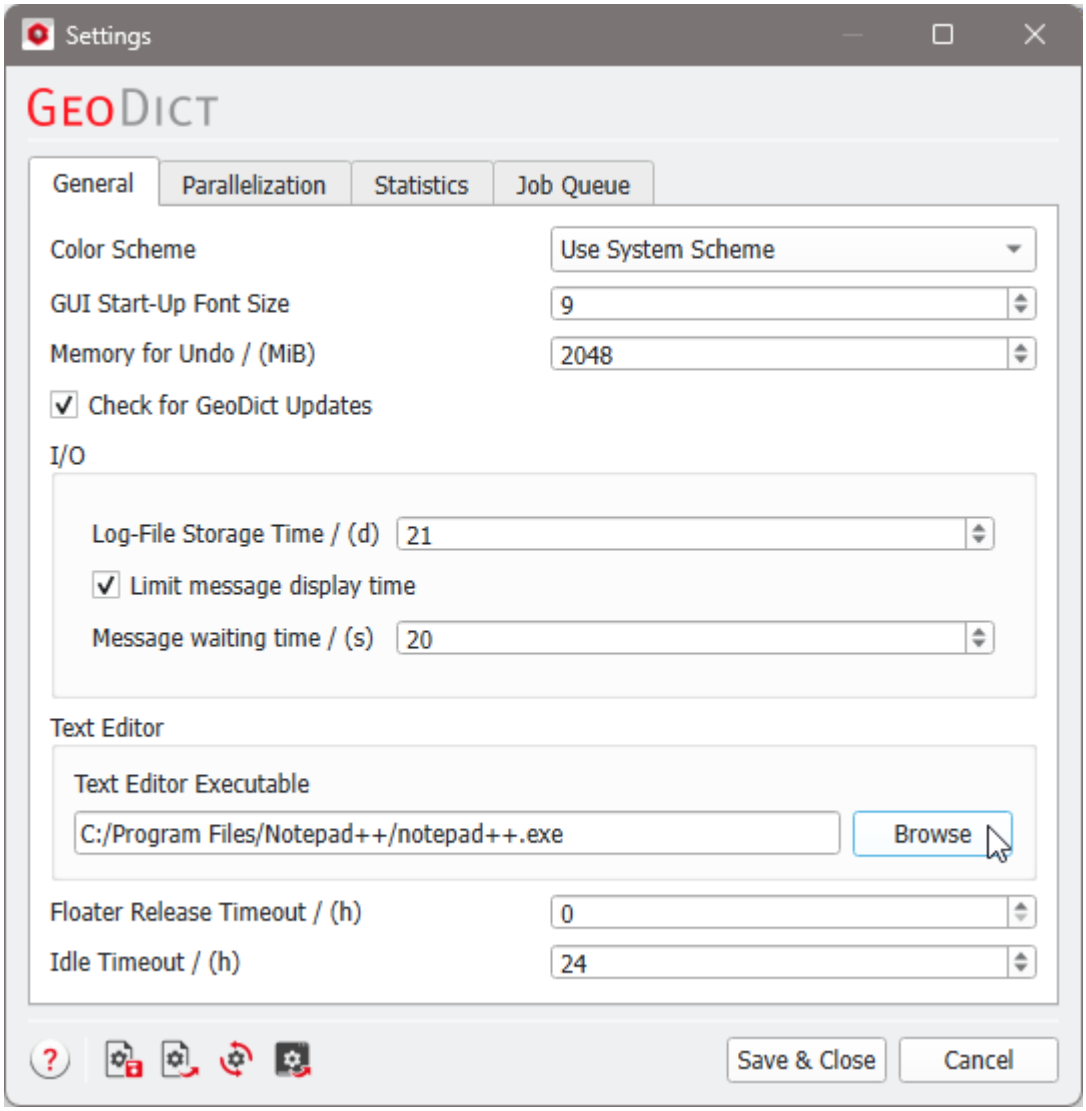
1.5 Choosing a Text Editor

You can define the text editor in which GeoDict opens macros when clicking **Edit** in the Macro Execution Control.

To define another text editor as the default text editor, open GeoDict and select **Settings** → **Settings ...** from the menu bar.



In the section **Text Editor**, at the bottom of the Settings dialog in the **General** tab, click **Browse** to find the path to the executable for the desired text editor.



Click **Save & Close** to apply the editor change. The choice is automatically saved into the start-up settings of GeoDict.

The next time the **Edit** button in the **Macro Execution Control** dialog box is clicked, the macro file is opened for editing in the selected text editor.

For other editors, browse for or enter the path to the desired editor.

Editors available for Windows

- **Notepad** is a simple text-editor provided during the installation of Windows. The Notepad text editor is called Editor in the Windows German edition. Syntax highlighting is not available and when opening files from other platforms (e.g., Linux), although the file is not corrupted, the commands are not displayed in easily readable lines.
- **WordPad**, another Windows built-in editor, is a good alternative for users who seldom edit macros. Files from Linux platforms are also displayed correctly. However, syntax highlighting is not available, and all formatting effects are removed when saving and closing the file. Files must be saved in .py and not in .py.rtf format.
- **Notepad++** is recommended. The free source code editor Notepad++ is the most comfortable alternative for Windows systems. Python syntax is highlighted and although there is no syntax highlighting for GMC macro files, their syntax is similar to C and HTML conventions and switching to C-syntax highlighting (Language ® C ® C++ in Notepad++ menu bar) helps improving readability of the files. You can also define his/her own syntax highlighting. Notepad++ is also included in the GeoDict-Tools installer.

Editors available for Linux

- **gedit** is provided with Ubuntu. Python syntax is highlighted.
- **Notepadqq** is the Linux version of Notepad++.
- **PyCharm** is not only an editor but an integrated development environment. While it can be very useful for experts, it is not recommended for beginners.

1.6 Parameter Studies

Using **parameter macros** is the smart choice when running studies in which some parameter values need to be combined with another parameter while both are varying.

For example, a simple macro, without variables, recorded while generating a fibrous structure with FiberGeo, can be modified to create a parameter macro containing variables. The introduced variables, random seed, object solid volume percentage (SVP) and fiber diameter, are used in combination to produce sequences of random realizations of the structure with a certain object solid volume percentage, i.e., series of structures are generated for every chosen SVP, while the SVP is gradually increased and the fiber diameter decreased.

Learn to create and run parameter studies:

- Create Parameter Macro 49
  Learn how to transform a simple macro into a parameter macro by adding variables. Run a parameter study using Vary Macro.
- Vary Parameters from GeoPy 56
  You can also run a macro from another macro, including parameter studies.
- Available Variable Types 58
  Find an overview of all available variable types.

1.6.1 Create Parameter Macro

Transform a simple macro into a parameter macro for a parameter study. For this, learn how to add variables to a recorded macro. Access these variables as parameters in the Macro Execution Control and run a parameter study using **Vary** parameters.

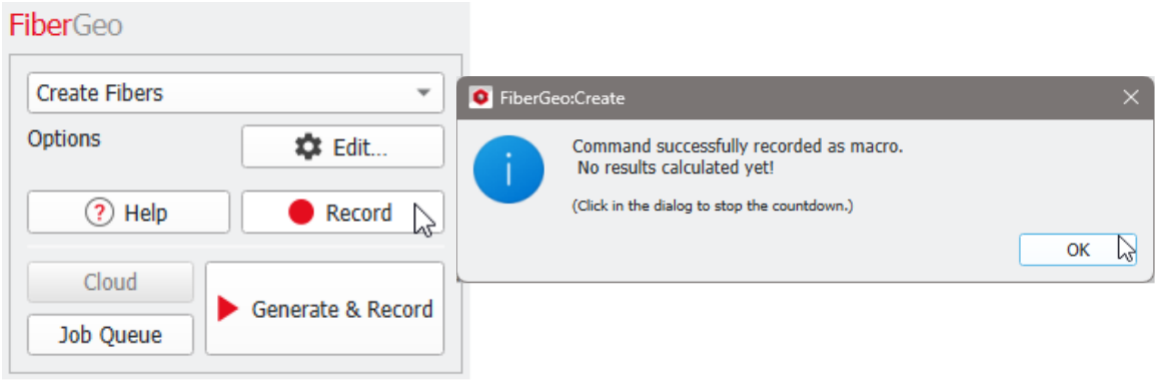
Note! Since GeoDict 2025 the variables block can be given as a list of dictionaries, one for each variable. This comes with the advantage, that it is not needed anymore to count the variables.

You can download the variable_study.py file created in this chapter for inspection in a zipped folder.

Download variable_study.zip

Record the macro

Start by recording the simple macro (simple_macro.py) during the generation of a fibrous structure with the default values in FiberGeo. Therefore, start macro recording 11. We chose the name "**simple_macro.py**". Then, select **Module** → **FiberGeo** and click **Record**. The default value for **Random Seed** is 42 and the single value for **Object Solid Volume Percentage** is 10.

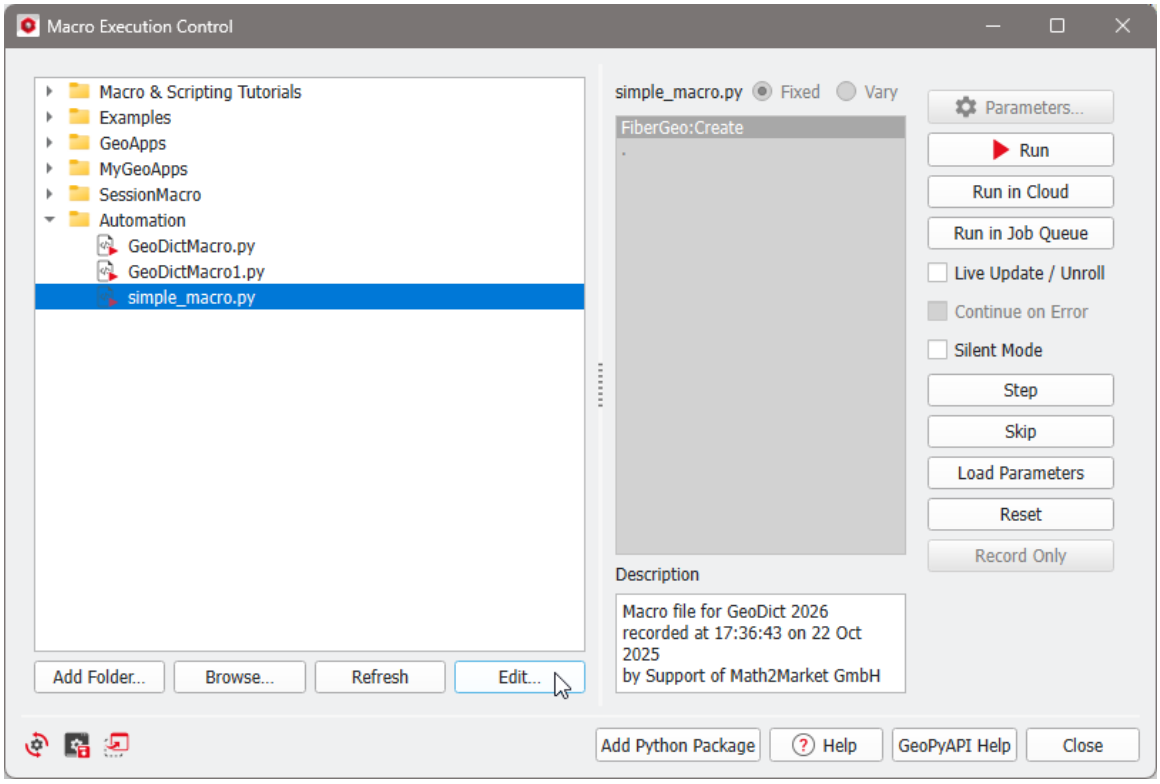


Afterwards, end the recording of the macro, by selecting **Macro → End Macro Recording**.

Check now **Macro → Execute Macro / Script ...**.

Click **Refresh** and, in the **Macro Execution Control** section, look for **simple_macro.py** in the pull-down menu list. The description area displays a short report about it.

simple_macro.py does not contain any variables at this point and thus, **Fixed** and **Vary** are grayed out.



Click **Edit...** and open **simple_macro.py** in the text editor of choice (here NotePad++).

No variables are yet defined in **simple_macro.py**. The **Variables** block is where they are defined and where they will be modified for the parameter study.

```
Variables = [
#   {
#     'Name'           : 'gd_SVP',
#     'Label'          : 'Solid Volume Percentage',
#     'Type'           : 'double',
#     'Unit'           : '%',
#     'ToolTip'        : 'Solid volume percentage of the created structure.',
#     'BuiltinDefault' : 10.0,
#     'Check'          : 'min0;max100'
#   },
  ]
```

The first command is to create a structure (FiberGeo:Create). In the parameter dictionary Create_args_1 first the **Domain** parameters are given. These parameters are not changed in our example.

Among other parameters, now follow the parameters corresponding to overlap mode, stopping criterion, number of objects, random seed, and other options that can be found in FiberGeo under the **Create Options** tab of the **FiberGeo Options** dialog.

From these parameters, the **Solid Volume Percentage**, the **Random Seed** and the **Fiber Diameter** will be used as variables and their entries in the macro are changed in the next step of this example.

Editing the macro

After opening it in a text editor, start editing the **simple_macro.py** by adding description information as shown here. This is later displayed in the description area of the **Macro Execution Control** section.

```
Description = '''
Parameter macro for a parameter study varying Solid Volume Percentage
and Random Seed in combination to generate random series of
increasingly dense fibrous structures with infinite circular fibers.
'''

Variables = [
   {
     'Name'           : 'gd_SVP',
     'Label'          : 'Solid Volume Percentage',
     'Type'           : 'double',
     'Unit'           : '%',
     'ToolTip'        : 'Solid volume percentage of the created
structure.',
     'BuiltinDefault' : 10.0,
     'Check'          : 'min0;max100'
   },
   {
     'Name'           : 'gd_RandomSeed',
     'Label'          : 'Random Seed',
     'Type'           : 'int',
     'Unit'           : '',
```

```
    'ToolTip'        : 'Random Seed of the created structure.',
    'BuiltinDefault' : 42,
  },
  {
    'Name'           : 'gd_FiberDiameter',
    'Label'          : 'Fiber Diameter',
    'Type'           : 'double',
    'Unit'           : 'µm',
    'ToolTip'        : 'Diameter of the created fibers.',
    'BuiltinDefault' : 10.0,
  },
]
```

In the **Variables** block, (as shown above) uncomment the variable block by deleting the # signs.

Copy and paste the default dictionary (the block from "**{**" to "**}**") to add a second and third variable element.

The first variable is given the Name **gd_SVP**, the second variable is given the name **gd_RandomSeed** and the third and last one is given the name **gd_FiberDiameter**. These names can be chosen as desired, but it is recommended to choose names describing their usage in the macro to improve readability. This is also the only reason for the prefix **gd_**, marking which variables in the macro are defined from the Parameters dialog and which are defined within the macro. The variables would also work without the prefix and different names, but then the macro code could be harder to understand for others.

The first and third variable are type **double** and the second is type **integer** ('**int**'). There are many more available variable types 58.

The starting BuiltinDefault values are **10** (%) for SVF, **42** for Random Seed and **10** (µm) for Fiber Diameter. Some helpful hints on syntax for these variables appear below the variables block.

To store the output of the parameter study, change from the project folder to a new folder with the name **Variable_Study**. For this purpose, add the **GeoDict:ChangeProjectFolder** command (before the FiberGeo command) to save the results in the new folder **'Variable_Study'**. This command can also be saved from the Session Macro 35 dialog, after changing the project folder in GeoDict.

```
ChangeProjectFolder_args = {
  'FolderName'          : 'Variable_Study',
  'CreateIfNotPresent'  : True,
}
gd.runCmd('GeoDict:ChangeProjectFolder', ChangeProjectFolder_args,
Header['Release'])
```

Right in the first line of the **Create_args** dictionary, change the **ResultFileName** from **'FiberGeo.gdr'** to:

f'FiberGeo_{gd_SVP}_{gd_RandomSeed}_{gd_FiberDiameter}.gdr',

to associate the name of the result files (in GDR format) to the outcome of the parameter study.

In this way, the result file names indicate the random seed, SVP and diameter values applied to the generated structure.

```
Create_args_1 = {
  'ResultFileName' : f'FiberGeo_{gd_SVP}_{gd_RandomSeed}
_{gd_FiberDiameter}.gdr',
```

In the next group of parameters, for **SolidVolumePercentage**, change the numerical value 10 to **gd_SVP** and, for **RandomSeed**, the value of 47 to **gd_RandomSeed**.

gd_SVP and **gd_RandomSeed** are placeholders for the sets of values to be defined when running the macro (**Macro Execution Control** dialog box).

```
  'NumberOfObjects'           : 100,
  'StoppingCriterion'         : 'SolidVolumePercentage',
  'SolidVolumePercentage'     : (gd_SVP, '%'),
  'Grammage'                  : (10, 'g/m^2'),

...

  'PercentageType'            : 0,
  'RandomSeed'                : gd_RandomSeed,
  'IsolationDistance'         : (0, 'm'),
  'MatrixDensity'             : (0, 'g/cm^3'),
```

Finally, in the block **Generator1**, more precisely in the subblock **DiameterDistribution** replace the Value 1e-05 by **gd_FiberDiameter * 1e-06**. The factor 1e-06 is needed, as the fiber diameters in the dictionary must be given in meter. Thus, the fiber diameter of the first fiber type can be changed in the parameter study, editing the value in microns.

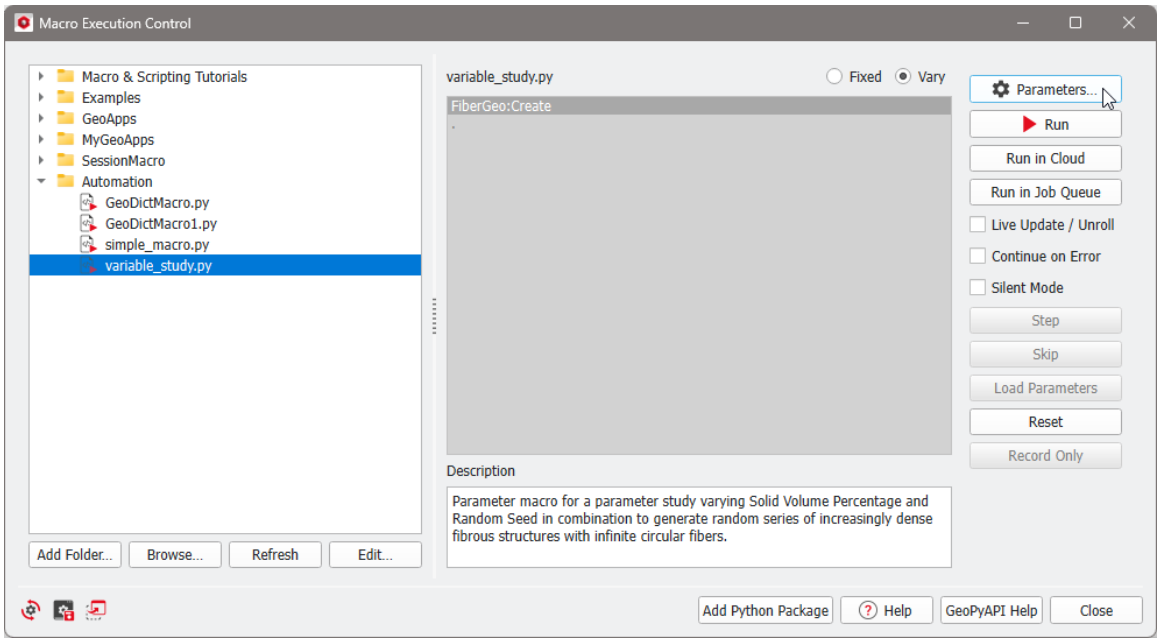
```
  'Generator1' : {
    'Material' : {
      'Type'        : 'Solid',
      'Name'        : 'Glass',
      'Information' : 'Fiber',
      },
    'Probability'               : (0.5, '1'),
    'SpecificWeight'            : (2.58, 'g/cm^3'),
    'Type'                      : 'InfiniteCircularFiberGenerator',
    'UseDTex'                   : False,
    'DiameterDistribution' : {
      'Type'  : 'Constant',
      'Value' : gd_FiberDiameter * 1e-06,
      },
```

In the editor, save the modified macro as **variable_study.py** (NotePad++: **File** → **Save As...**).

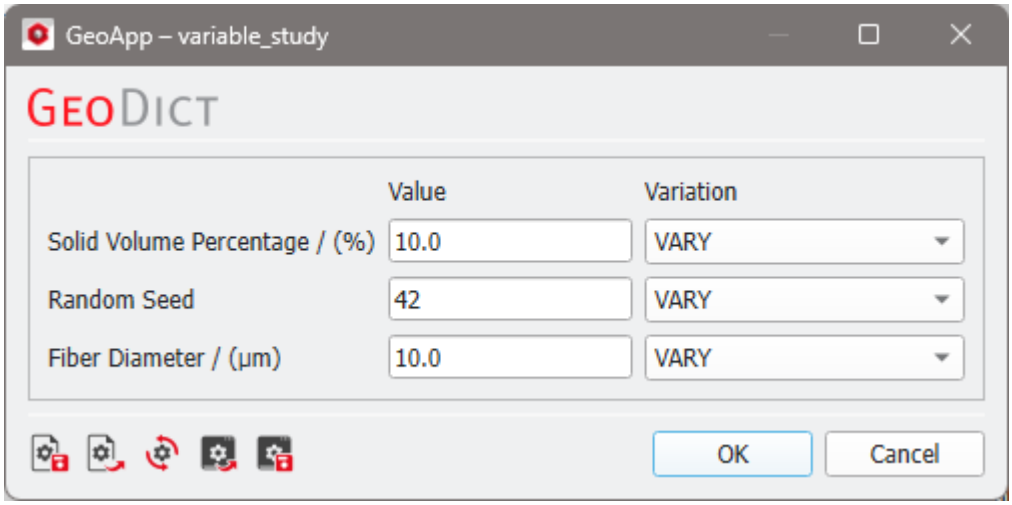
Run the parameter study

Back in the **Macro Execution Control** section, click **Refresh** to actualize the left panel and select (the just saved) **variable_study.py** from it.

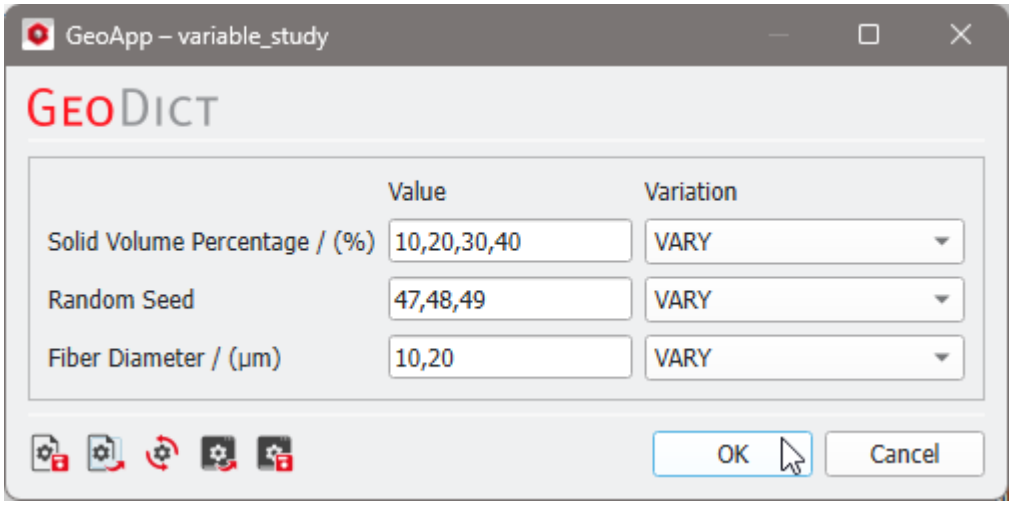
The text entered under **Description** in the edited macro is shown in the description area and, since now the macro contains variables, **Vary** is available to be checked. Check it and click the **Parameters** button.



The **BuiltinDefault** values that were specified in the variables block (10, 42 and 10) appear in the boxes for **Solid Volume Percentage**, **Random Seed** and **Fiber Diameter**. The labels of both variables have been taken from the **variable_study.py** file.



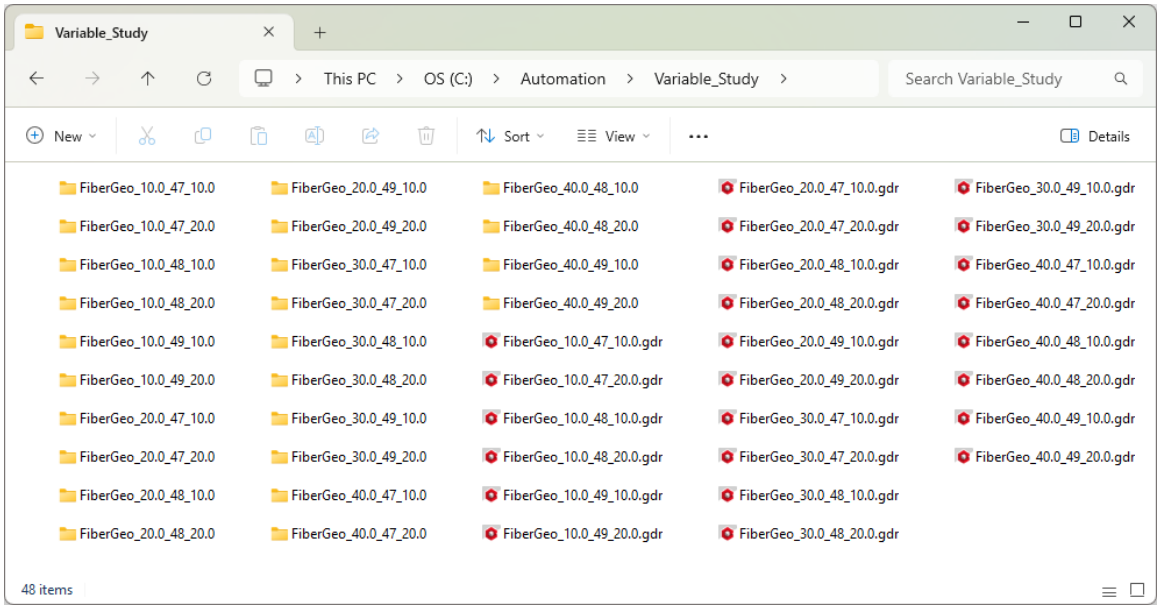
To set the parameter study, enter four values of increasing **Solid Volume Percentage** (10%, 20%, 30% and 40% SVP), three random seed values (e.g., 47, 48, and 49) and two values for **Fiber Diameter** (e.g., 10 and 20). Leave the **Variation** for all three at **VARY**.



Click **OK** and, in the **Macro Execution Control** section, click **Run**.

The execution of the **variable_study.py** macro takes only a short time and creates three random realizations of a structure for every one of the four SVP values, combined with every fiber diameter value.

The outcome is 48 items saved in the project folder Variable_Study: 24 result files (e.g., FiberGeo_10.0_47_10.0.gdr) and 24 folders, each with a structure file (*.gdt) inside (e.g., FiberGeo_10.0_47_10.0).



These 24 result files can be opened in GeoDict, and the Result Viewer offers the possibility to combine some or all results in a plot. See the Result Viewer topic for more details.

1.6.2 Vary Parameters from GeoPy

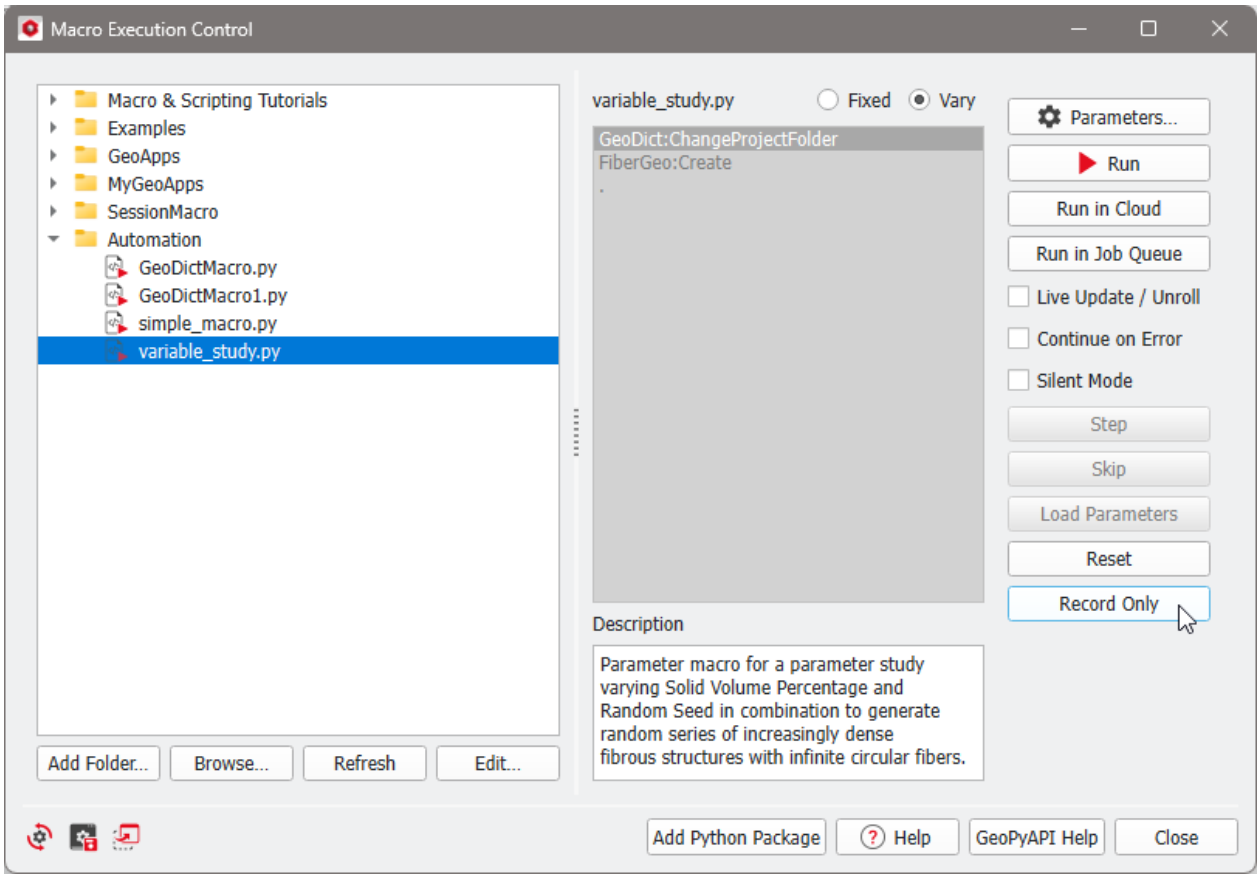
Having transformed a simple macro to a parameter macro it is possible to automate the parameter study in the Python macro. For this, start macro recording as described in the topic macro recording 11.

You can download the run-variable-study.py file created in this chapter for inspection in a zipped folder.

Download run-variable-study.zip

Open the **Macro Execution Control**, check **Vary** and edit the parameters 54 for the variable study.

Click **Record Only** to save the **GeoDictMacro:VaryPython** command without running the macro.



Click **OK** in the opening dialog.

The recording of the macro is stopped by selecting **Macro → End Macro Recording.**

In the Macro Execution Control click **Refresh**, highlight the new Python macro and Click **Edit**.

The **GeoDictMacro:VaryPython** command is located after the **Variables** section. This command can be used for any parameter macro. The file path and the variables have to be given. The entries in the Variables dictionary correspond to the vary parameters dialog box.

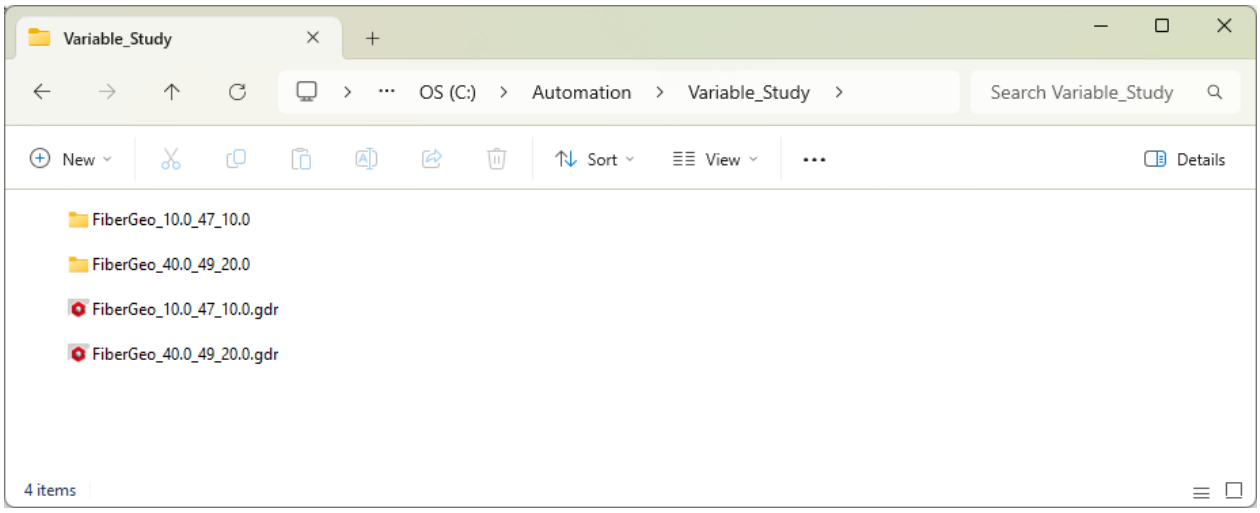
```
VaryPython_args_1 = {
  'FileName'        : 'C:/Automation/variable_study.py',
  'ContinueOnError' : False,
  'Variables' : {
    'gd_SVP' : {
      'ValueList' : ([10, 20, 30, 40], '%'),
      'Variation' : 'VARY',
      },
    'gd_RandomSeed' : {
      'ValueList' : [47, 48, 49],
      'Variation' : 'VARY',
      },
    'gd_FiberDiameter' : {
      'ValueList' : ([10, 20], 'µm'),
      'Variation' : 'VARY',
      },
    },
  }
gd.runCmd("GeoDictMacro:VaryPython", VaryPython_args_1, Header['Release'])
```

For example, the value lists can be changed so that the number of the list entries is the same. Thus, the **'Variation'** of gd_RandomSeed and gd_FiberDiameter can be changed from **'VARY'** to **'gd_SVP'**.

```
VaryPython_args_1 = {
  'FileName'        : 'C:/Automation/variable_study.py',
  'ContinueOnError' : False,
  'Variables' : {
    'gd_SVP' : {
      'ValueList' : ([10, 40], '%'),
      'Variation' : 'VARY',
      },
    'gd_RandomSeed' : {
      'ValueList' : [47, 49],
      'Variation' : 'gd_SVP',
      },
    'gd_FiberDiameter' : {
```

```
        'ValueList' : ([10, 20], 'µm'),
        'Variation' : 'gd_SVP',
        },
      },
    }
gd.runCmd("GeoDictMacro:VaryPython", VaryPython_args_1, Header['Release'])
```

After saving the macro click **Run** in the **Macro Execution Control** and the resulting folder Variable_Study only contains two result files and two result folders.



1.6.3 Available Variable Types

The variables block in GeoDict Python macros provides many options. A summary of all these options and some short explanations and examples can be found in the comment block after the variables block in a recorded macro.

```
Variables = [
#   {
#     'Name'           : 'gd_SVP',
#     'Label'          : 'Solid Volume Percentage',
#     'Type'           : 'double',
#     'Unit'           : '%',
#     'ToolTip'        : 'Solid volume percentage of the created structure.',
#     'BuiltinDefault' : 10.0,
#     'Check'          : 'min0;max100'
#   },
  ]

# Explanations of variables syntax:
###################################
# Name:           mandatory, name of the variable by that it can be addressed in the macro, must not contain
white spaces!
# Label:          optional,  appears as text in the GeoDict GUI. If not present, then Name is used also as
Label
#                            for type gdrstring, this is ignored and the default label is always shown
# Type:           mandatory, known types are bool, boolgroup, double, uint, int, string, gdrstring,
filestring, folderstring, material, combo, table, combogroup, labelgroup
# Unit:           optional,  appears only in GUI (not used to rescale any input parameters automatically)
#                            for type gdrstring,  Unit is ignored
#                            for type filestring, Unit contains the file suffix
#                            for type material,   Unit must be solid, fluid or porous
#                            for type combo,      Unit must contain the possible string-values for the
variable separated by semicolon
#                            for type table,      Unit must be a list of type strings, allowed is "int",
"float", "string". E.g. ["int", "float", "string"] for three columns.
# ToolTip:        optional,  appears in GUI (must be in one line)
# BuiltinDefault: optional,  default value which is used in macro (if not given, defaults to 0 or empty
string)
#                            for type table, this should be a python list of entries, left to right, top to
bottom, e.g. [1,2.0,"three"].
# ColumnHeaders:  optional,  only valid for type table: List of header texts for each table column, e.g.
["Column 1", "Second column", "Third Column"]
# Check:          optional,  known checks are positive, negative, min, max (checks are separated by semicolon)
# Member:         optional,  defines the member of group type variables. For Labelgroups defined by a list,
for combogroup and boolgroup defined by a dictionary that maps states to lists
```

The variables block defines the parameters displayed in the **Parameters** dialog in the Macro Execution Control 22.

In the following, the available types of variables are described, and examples are given. The type must be given as a string for the key **'Type'**.

int

For a variable of type **'int'** only integer values are allowed, i.e., ... -2, -1, 0, 1, 2, ... If **Vary** is checked in the **Macro Execution Control**, also lists of values can be entered with the start:step:end 27 syntax.

```
Variables = [{
   'Name'  : 'gd_int',
   'Label' : 'Variable',
   'Type'  : 'int'}]
```

uint

For a variable of type **'uint'** only nonnegative integer values are allowed for this variable, i.e., 0, 1, 2, ... In the **Parameters** dialog it is also possible to change the value by clicking the arrows on the right or by turning the mouse wheel while the cursor is

rested on the parameter box. If a check is added, the arrows will not allow values outside the given range. If **Vary** is selected in the **Macro Execution Control**, also lists of values can be entered with the start:step:end syntax.

```
Variables = [{
  'Name'  : 'gd_uint',
  'Label' : 'Variable',
  'Type'  : 'uint'}]
```

double

For a variable of type **'double'** any floating point number is allowed, e.g., -0.75, 10.3, 42.999. If **Vary** is checked in the **Macro Execution Control**, also lists of values can be entered with the start:step:end syntax.

```
Variables = [{
  'Name'  : 'gd_doublel',
  'Label' : 'Variable',
  'Type'  : 'double'}]
```

bool

A **'bool'** variable defines a checkbox in the **Parameters** dialog. Possible values for the optional key **'BuiltinDefault'** are **False** (not checked) and **True** (checked).

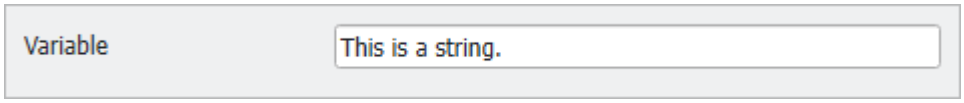
```
Variables = [{
  'Name'  : 'gd_bool',
  'Label' : 'Variable',
  'Type'  : 'bool'}]
```

string

Everything typed in the parameter box for a variable of type **'string'** will be handled as a string in the macro.

```
Variables = [{
  'Name'  : 'gd_string',
  'Label' : 'Variable',
```

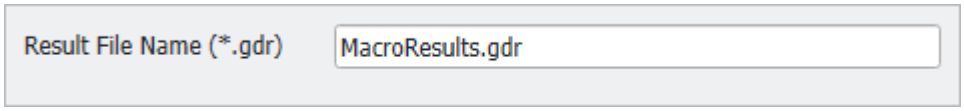
```
  'Type'   : 'string'}]
```



gdrstring

For a variable of type **'gdrstring'** enter a string in the parameter dialog. If not already given, the file extension .gdr is automatically added when clicking OK or somewhere in the Parameters dialog. Use this variable type for naming the result file of your macro. The label is always Result File Name. Given labels are ignored.

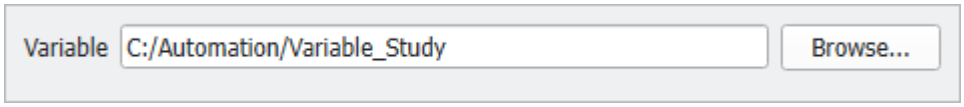
```
Variables = [{
  'Name'   : 'gd_resultfile',
  'Type'   : 'gdrstring'}]
```



folderstring

For a variable of type **'folderstring'** in the **Parameters** dialog a **Browse** button will appear next to the parameter box to search for the desired folder on the computer.

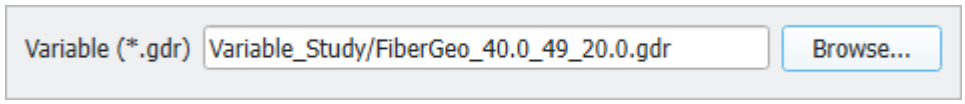
```
Variables = [{
  'Name'   : 'gd_folder',
  'Label'  : 'Variable',
  'Type'   : 'folderstring'}]
```



filestring

For a variable of type **'filestring'** in the parameter dialog a Browse button will appear next to the parameter box to search for the desired file on the computer. The **'Unit'** is the file extension for this variable type and must be specified, e.g., 'gdr' or 'xlsx'. You can also enter different file types by semicolon, e.g., "gdr;gdt".

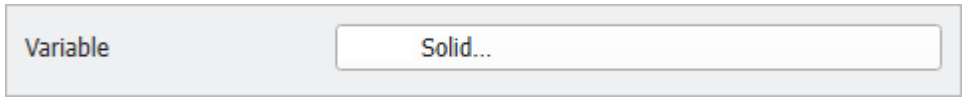
```
Variables = [{
  'Name'   : 'gd_file',
  'Label'  : 'Variable',
  'Type'   : 'filestring',
  'Unit'   : 'gdr'}]
```



material

For a variable of type **'material'** the desired material can be selected from the GeoDict material data base. The **'Unit'** must be specified as **'solid'**, **'fluid'** or **'porous'**. Also, a **'BuiltinDefault'** value is needed, e.g., 'Manual'.

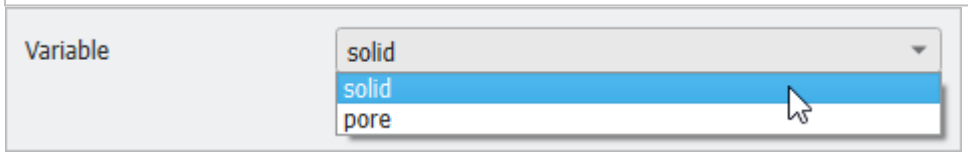
```
Variables = [{
  'Name'           : 'gd_material',
  'Label'          : 'Variable',
  'Type'           : 'material',
  'Unit'           : 'solid',
  'BuiltinDefault' : 'Manual'}]
```



combo

A variable of type **'combo'** defines a value choice, that will be displayed in a pull-down menu (also named combo box) in the parameter dialog. For **'Unit'** define a string with the components separated by semicolon. The **'BuiltinDefault'** value defines, which combo box entry is selected by default. If not given, a warning appears and the first entry is selected.

```
Variables = [{
  'Name'           : 'gd_combo',
  'Label          ': 'Variable',
  'Type'           : 'combo',
  'Unit'           : 'solid;pore',
  'BuiltinDefault' : 'solid'}]
```



table

A variable of type **'table'** will transform the values entered in the **Parameters** dialog into a list. The number of columns is defined with the key **'Unit'**. There, the types for the different columns must be given as a list. Available column types are **'int'**, **'float'** and **'string'**. The column headers are also given as a list of strings and are optional.

In the **Parameters** dialog a new row is added as soon as at least one value is entered in each existing row.

In the following example, three columns are given. Here, the values in the first column must be integers, the values in the second column float and the values in the third column string, as defined for the key **'Unit'**. The **'BuiltinDefault'** values define two rows in the table.

```
Variables = [{
  'Name'           : 'gd_table',
  'Label'          : 'Variable',
  'Type'           : 'table',
  'Unit'           : ['int','float','string'],
  'ColumnHeaders'  : ['column 1', 'column 2', 'column3'],
  'BuiltinDefault': [1, 2.0, 'three', 4, 5.0, 'six']
}]
```

Variable

	column 1	column 2	column3
1	1	2.0	three
2	4	5.0	six
3			

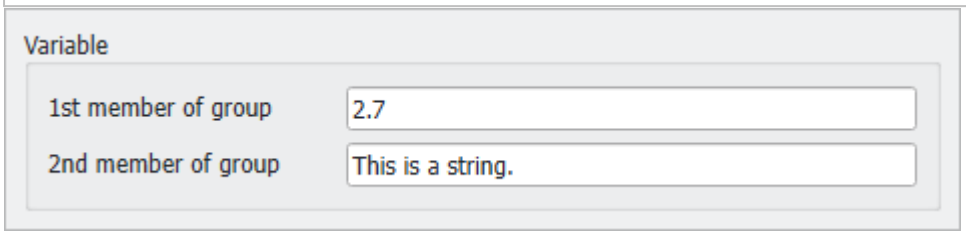
labelgroup

A variable of type **'labelgroup'** defines a group within the **Parameters** dialog. The key **'Member'** is mandatory and defines which of the following variables will belong to the group. The members have to be given in a list, containing the members name as a string. The **'BuiltinDefault'** must be **True**. The members are defined separately as variables and can have any type.

In the following example, a group with two members is defined in **'Variable1'**. The first member is defined as **'Variable2'** as type **'double'** and the second member is defined as **'Variable3'** as type **'string'**. Their names **'member1'** and **'member2'** are given in the list for the key **'Member'**.

```
Variables = [{
  'Name'           : 'gd_labelgroup',
  'Label'          : 'Variable',
  'Type'           : 'labelgroup',
  'Member'         : ['member1', 'member2'],
  'BuiltinDefault': True},
 {
  'Name'           : 'member1',
  'Label'          : '1st member of group',
```

```
  'Type'           : 'double'},
 {
  'Name'           : 'member2',
  'Label'          : '2nd member of group',
  'Type'           : 'string'}]
```

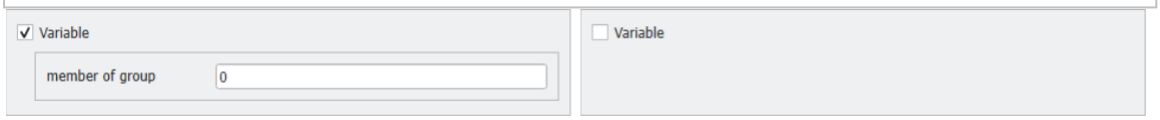


boolgroup

A variable of type **'boolgroup'** defines two groups within the **Parameters** dialog. Checking or not checking the checkbox decides which group is shown. The members have to be defined as separate variables and can have any type. The names must be given for the key **'Member'** for the boolgroup variable, as a dictionary, consisting of the keys **'true'** and **'false'** and the respective group members as a list.

In the following example, only one group is defined. This results in an empty group if the checkbox is not checked, corresponding to the not given value **'false'**.

```
Variables = [{
  'Name'           : 'gd_boolgroup',
  'Label'          : 'Variable',
  'Type'           : 'boolgroup',
  'Member'         : {'true' : ['member']},
  'BuiltinDefault': True},
 {
  'Name'           : 'member',
  'Label'          : 'member of group',
  'Type'           : 'double'}]
```



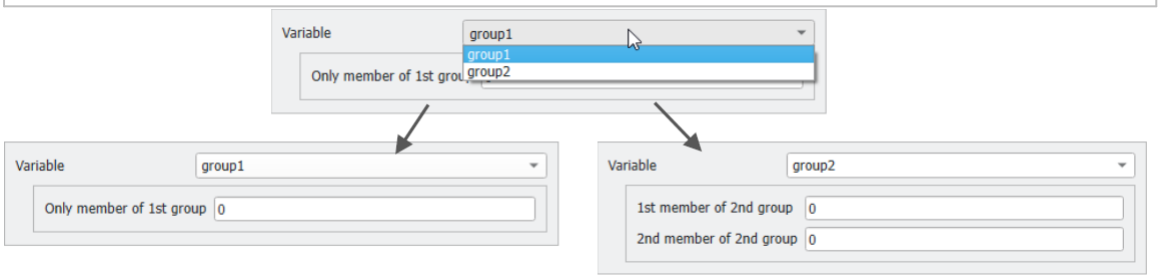
combogroup

A variable of type **'combogroup'** defines multiple groups. The selection from the pull-down menu decides which group is displayed. The list defining the content of the pull-down menu must be defined for the key **'Unit'**. The values must be given as a string, values separated by comma. The members of the groups must be defined as separate variables and can have any type. The names must be given for the key **'Member'** for the combogroup variable, as a dictionary, consisting of the defined keys (values in the pull-down menu, defined in **'Unit'**) and the respective group members as a list. The

'BuiltinDefault' value defines which combo box entry is selected by default. If not given, a warning message appears and the first entry is used.

In the following example two groups can be selected. Observe how the available parameters change according to the selected group in the **Parameters** dialog.

```
Variables = [{
  'Name'    : 'gd_combogroup',
  'Label'   : 'Variable',
  'Type'    : 'combogroup',
  'Unit'    : 'group1;group2',
  'Member'  : {'group1' : ['onlymemb
  'BuiltinDefault' : 'group1'},
 {
  'Name'    : 'onlymember',
  'Label'   : 'Only member of 1st gr
  'Type'    : 'double'},
 {
  'Name'    : 'member1',
  'Label'   : '1st member of 2nd gro
  'Type'    : 'int'},
 {
  'Name'    : 'member2',
  'Label'   : '2nd member of 2nd gro
  'Type'    : 'int'}]
```



1.7 Advanced Scripting

GeoDict supports Python scripting. By selecting **Macro → Execute Macro/Script...** a *.py file can be selected and then executed by a built-in **Python 3.11** interpreter. All of the Python standard library should be usable from within a Python macro. In addition, you can use the API (Application Programming Interface) for GeoDict specific functions and packages to access GeoDict files.

A very helpful official Python tutorial can be found at https://docs.python.org/3.11/tutorial/.

Python scripts are executed as described in Execute Macro / Script 12 for GeoPy macros.

Advanced scripting possibilities:

- GeoDict API 66
  Use the GeoDict Application Programming Interface everywhere in a Python macro for GeoDict specific applications.
- Shipped Python Modules 129
  There are many Python packages shipped with GeoDict.
- PowerPoint Report Generation 131
  Automize PowerPoint report generation from your GeoDict results using our simplified wrapper API.
- Access to Result Files 137
  Access and edit GeoDict result files (*.gdr).
- Create Custom Result Files 142
  Create custom GeoDict result files, for example if developing your own GeoApp.
- Access to 3D Result Files 159
  Access the 3D result files, as for example structures or volume fields.
- Error Reporting 167
  Learn about common error messages to debug your Python scripts.

1.7.1 GeoDict API

The special object called **gd** is available everywhere within a Python macro. The whole GeoDict Application Programming Interface (**API**) is exposed via the **gd**-object. In the following, the methods provided by the built-in **gd**-object are documented. The interface allows running any GeoDict command that a macro can execute.

The GeoPy API covers the following topics:

- GeoDict Command Functions 67
  Work with GeoDict commands.
- Structure Functions 70
  Access and edit structure files (*.gdt).
- GAD-Object Functions 78

The GeoPy API covers the following topics:

Access and edit GAD-objects.

- Result Functions 80
  Access and edit volume fields, triangles and result files.
- Material Functions 85
  Access material information.
- GeoDict Settings Functions 87
  Get information about GeoDict settings, as release version or location of different folders.
- Macro Functions 90
  Get information about the current macro.
- Progress Bar Functions 91
  Set-up a progress bar for your macro.
- Pop-Up Functions 95
  Prevent or raise pop-ups.
- Dialog Class 97
  Create your own customized dialog for variable input or messages.
- Graph Dialog Class 116
  Create a GeoDict graph with the graph dialog class.
- Visualization Functions 118
  Get the current view status as dictionary or plot.
- ImportGeo-Vol Functions 120
  ImportGeo-Vol specific functions to create your own custom image processing filter.
- Particle Functions 123
  Access the data corresponding to loaded particles from FilterDict or AddiDict simulations.

GeoDict Command Functions

All actions performed in GeoDict are commands, executable from Python with some variations. Also, the parameter settings of the commands can be read from Python.

gd.runCmd

This allows to run any GeoDict command that a macro can execute.

For commands that produce GDR files, the function returns the name of the generated file, which can be different from the name specified if a file of the same name did already exist, e.g., "PoreSizes_no1.gdr". It is therefore recommended to use the returned file name when analyzing the results.

Input:

- command name, string

- This is the name of the command as they appear in the Session Macro 35 dialog, e.g., "GeoDict:LoadFile" to load a GDT file.

- parameters, dictionary
  - This is a Python dictionary holding the arguments as described in Structure of a GeoPy macro 8.

- version, string
  - GeoDict version for which this macro was written, e.g., "2026". Usually it is given as Header["Release"], taking the release from the header 7.

Example:

In the following example, the function is used to terminate GeoDict. For other examples, see also below under the getViewStatus 118 or the getBuiltinDefaults 69 command.

```
gd.runCmd("GeoDi  # terminates GeoDict, the dictionary
ct:Terminate",    # for this command is empty
{}, "2026")
```

gd.runCmdIgnoreExtraKeys

Works similar to gd.runCmd 67, but ignores unnecessary keys in the Python dictionary of the command.

Input:

- command name, string
  - This is the name of the command as they appear in the Session Macro 35 dialog, e.g., "GeoDict:LoadFile" to load a GDT file.

- parameters, dictionary
  - This is a Python dictionary holding the arguments as described in Structure of a GeoPy macro 8.

- version, string
  - GeoDict version for which this macro was written, e.g., "2026".

gd.runCmdFromGPS

Executes a command from a *.gps file, that can be obtained directly from a dialog. The command has no return value.

Input:

- gps file path, string

Example:

If the desired settings for a fiber structure are saved from the **FiberGeo Create Options** dialog into a *.gps file with the name FiberGeo.gps, the fiber structure can be created with this command:

```
gd.runCmdFromGPS("FiberGeo.gps")     # generates a fiber structure from
                                     a *.gps file
```

gd.getBuiltinDefaults

Returns the built-in default argument dictionary for a command. This can then be modified and passed to gd.runCmd 67.

Input:

- command name, string
  - This is the name of the command as they appear in the Session Macro 35 dialog, e.g., "GeoDict:LoadFile" to load a GDT file.
- version, string (optional)
  - GeoDict version for which this macro was written, e.g., "2026".

Example:

```
Create_args =                        # get the arguments for
gd.getBuiltinDefaults("FiberGeo:Cr   # "FiberGeo:Create"
eate")

Create_args['SolidVolumePercentage   # change solid volume
'] = 20                              # fraction to 20%

gd.runCmd("FiberGeo:Create",         # run the fiber generation
Create_args, '2026')
```

gd.getCurrentSettings

Returns the current settings argument dictionary for a command. This can then be modified and passed to gd.runCmd 67.

Input:

- command name, string
  - This is the name of the command as they appear in the Session Macro 35 dialog, e.g., "GeoDict:LoadFile" to load a GDT file.
- version, string (optional)
  - GeoDict version for which this macro was written, e.g., "2026".

Example:

```
Create_args =                          # get the arguments for
gd.getCurrentSettings("FiberGeo:Cr     # "FiberGeo:Create"
eate")

Create_args['SolidVolumePercentage     # change solid volume
']=(20, '%')                           # fraction to 20%

gd.runCmd("FiberGeo:Create",           # run fiber generation
Create_args, '2026')
```

gd.setCurrentSettings

Sets the settings for the given GeoDict command in the corresponding options dialog.

Input:

- command name, string
  - This is the name of the command as they appear in the Session Macro 35 dialog, e.g., "GeoDict:LoadFile" to load a GDT file.
- parameters, dictionary
  - This is a Python dictionary holding the arguments as described in Structure of a GeoPy macro 8.
- version, string (optional)
  - GeoDict version for which this macro was written, e.g., "2026".

Example:

```
Create_args =                          # get the arguments for
gd.getCurrentSettings("FiberGeo:Cr     # "FiberGeo:Create"
eate")

Create_args['SolidVolumePercentage     # change solid volume
']=(20, '%')                           # fraction to 20%

gd.setCurrentSettings("FiberGeo:Cr     # set the changed settings
eate", Create_args, "2026")            # in the FiberGeo Options
                                       # dialog
```

Structure Functions

Get information about the currently loaded structure file, as for example size, voxel length, solid volume percentage and many more. Edit the structure and set the structure in GeoDict.

gd.getDomain

Gets the settings of the domain of the currently loaded geometry and returns it as a Python dictionary. This dictionary contains information about domain size, origin, voxel

length, material of ID00, periodicity and overlap settings, if available.

Input:

- version, string (optional)
  - GeoDict version for which this macro was written, e.g., "2026".

Example:

```
domain = gd.getDomain('2026')          # gets the domain parameters
                                       # as a dictionary

nx = domain['Domain']['NX']            # assigns the number of
                                       # voxels in x-direction
                                       # to the variable nx

vl = domain['Domain']                  # assigns the voxel length to
['VoxelLength']                        # variable vl

material = domain['Domain']            # assigns the material name
['Material']['Name']                   # of material ID00
                                       # to variable material

print(f'The loaded structure has       # show message in the console
{nx} voxels in X-direction, a
voxel length of {vl} and contains
the material {material}.')
```

gd.getVolDimensions

Returns a 3-tuple (nx,ny,nz) containing the size of the currently loaded geometry in number of voxels. Returns **None** if no geometry is present. This command can be assigned to individual variables in Python using tuple deconstructions as shown in the example.

Input: None

Example:

```
nx, ny, nz = gd.getVolDimensions()  # assigns the number of voxels in
                                    # x-direction to the variable nx,
                                    # and the number of voxels in
                                    # y- and z-direction to ny and
                                    nz,
                                    # respectively
```

gd.getVoxelLength

Returns the voxel length of the current structure in meters.

Input: None

Example:

```
vl = gd.getVoxelLength()                # assigns the voxel length to the
                                        variable vl
```

gd.getVoxelCounts2D

Returns a list of the slice-wise voxel counts in the given direction for the given material ID. Returns **None** if no geometry is present.

Input:

- direction, integer
- material index, integer

Example:

```
voxelCounts =                           # get the number of voxels
gd.getVoxelCounts2D(2,1)                # of material ID 1 in
                                        # Z-direction for each slice
                                        # and assign them
                                        # as a list to the variable
                                        # voxelCounts

ZSlice_5 = voxelCounts[4]               # compute the total number
                                        # of voxels in the structure
                                        # and assign it to the
                                        # variable TotalVoxels

gd.msgBox(f"In slice 5 {ZSlice_5}       # show message dialog
voxels have MaterialID 1.")
```

gd.getVoxelCounts3D

Returns a 256-element list of voxel counts for each color (material index) for the currently loaded geometry. Returns **None** if no geometry is present.

Input: None

Example:

```
nx , ny, nz =                           # get the number of
gd.getVolDimensions()                   # voxels in all three
                                        # directions and assign
                                        # them to variables

TotalVoxels = nx * ny * nz              # compute the total
                                        # number of voxels
                                        # in the structure
                                        # and assign it to
                                        # the variable
                                        # TotalVoxels
```

```
Voxels = gd.getVoxelCounts3D()          # gets list of voxel
                                        # counts for material IDs

ID_1 = Voxels[1]/TotalVoxels * 100      # computes volume
                                        # percentage of material
                                        # ID 1 and assign it to
                                        # variable ID_1

gd.msgBox(f"MaterialID 1 is             # show message dialog
assigned to {ID_1}% of the              # of result
structure.")
```

gd.getSolidVolumeFraction

Returns the solid volume fraction of the currently loaded geometry.

Input: None

Example:

```
svp = gd.getSolidVolumeFraction()    # compute solid volume percentage
* 100

gd.msgBox(f"The SVP is {svp}%.")     # show message dialog of result
```

gd.getSelectedVoxels

Returns the positions of the currently selected voxels as a list of tuples (x,y,z). Note, that the positions returned with this command are not exactly the same, as given in the GUI. That is because the positions count starts with (0,0,0) for the command getSelectedVoxels and with (1,1,1) for the GUI.

Input: None

Example:

For the following example a structure has to be loaded and one or more voxels must be selected:

```
Voxels = gd.getSelectedVoxels()      # assign list of
                                     # selected voxels to
                                     # variable Voxels

gd.msgBox(f"The first selected       # show message dialog
voxel is located at position
{Voxels[0]}")
```

gd.getStructure

Returns the currently loaded structure as a 3D 8-bit numpy array. Each entry corresponds to a voxel and contains its material ID (0-255).

Input: None

Example:

The following example writes the currently loaded structure into a *.csv file, where the first row contains the volume dimensions nx, ny and nz, followed by rows each containing the voxel values along a single Z-row.

```
with open("Structure.csv", "w") as fd:                  # open output file for writing
                                                        # (create new file with the
                                                        # given name, if file does not exist)
                                                        # and assign it to fd.
                                                        # The file stays open for the
                                                        # following indented section.

  Structure = gd.getStructure()                         # assign 3D numpy array of
                                                        # currently loaded structure to
                                                        # variable Structure. data type
                                                        # is 8-bit unsigned (uint8)

  nx, ny, nz = gd.getVolDimensions()                    # assign structure volume dimensions
                                                        # to variables nx, ny and nz

  fd.write(f"{nx},{ny},{nz}\n")                         # write dimensions of volume
                                                        # in first row

  for x in range(nx):                                   # loop over all x-coordinates

    for y in range(ny):                                 # loop over all y-coordinates

      row = Structure[x,y,:]                            # assign the z-row with
                                                        # x-coordinate x and y-coordinate y
                                                        # to the variable row

      strList = [f"{voxel_value}" for voxel_value in row]   # transform all entries of the row
                                                            # in strings and write
                                                            # them in the string list strList

      fd.write(",".join(strList) + "\n")                # writes all entries of strList
                                                        # in the csv file, separated
                                                        # by commas, adds a new line
                                                        # at the end of the list
```

For example, the 3d numpy array `[[[2, 1], [4, 3]], [[7, 5], [8, 6]]]` of a 2x2x2 structure is written into a csv file structured as follows:

```
2,2,2
2,1
4,3
7,5
8,6
```

gd.setStructure

This command has no return value but takes a 3D numpy array containing values between 0 and 255, defining the material ID of the described voxel, and sets it as GeoDict's current structure. This causes volume fields to be unloaded. The function is often used in combination with **gd.getStructure**, 73 which returns the current structure as a 3D numpy array. Using numpy or list operations, the structure can be manipulated and finally the new structure can be loaded to GeoDict via gd.setStructure.

Input:

- material values for structure, 3D numpy array
- voxel length, float

Example:

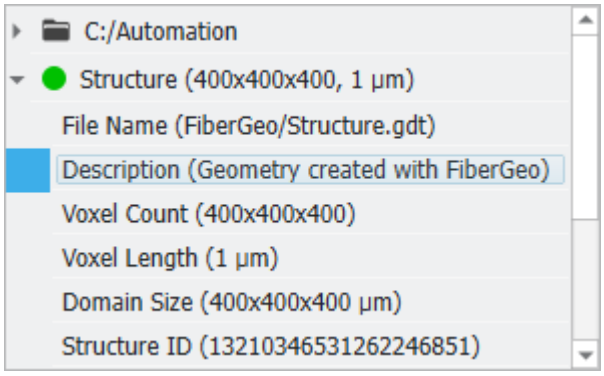
For example, if a 3D structure is saved as a *.csv file, structured in the same way as in the example for **gd.getStructure** 73 above, this structure can be visualized in GeoDict with the **gd.setStructure** command:

```
import numpy as np                      # import Python module numpy
                                        # to create numpy arrays

with open("Structure.csv", "r") as      # open input file for reading
fd:                                     # and assign it to fd.
                                        # The file is closed after the
                                        # last indented line following

  first_row =                           # read first row and
fd.readline().strip()                   # remove newline \n

  first_row_list =                      # assign list of first_row
first_row.split(",")                    # entries splitted by commas
                                        # to variable first_row_list

  nx, ny, nz =                          # assign the volume dimensions
int(first_row_list[0]),                 # contained in the first row
int(first_row_list[1]),                 # to variables nx, ny and nz
int(first_row_list[2])

  voxel_value_list = []                 # an empty list is assigned to
                                        # variable voxel_value_list to
                                        # store integer values of all
                                        voxels
```

```
  for line in fd:                       # loop over all lines in the
                                        # *.csv file, starting with the
                                        # second row, as the first
                                        # was already read

    line_stripped = line.strip()        # remove whitespace before
                                        # and after line. in this case,
                                        # remove newline at end of line.

    LineList =                          # assign a list of all entries
line_stripped.split(",")                # from line separated by
                                        # commas to variable LineList

    LineList = [int(x) for x in         # convert each voxel_value
LineList]                               # string to an integer number

    voxel_value_list += LineList        # append voxel values of this
                                        # row to list

  voxel_values =                        # convert voxel values to numpy
np.array(voxel_value_list,              # array. data type needs to
dtype=np.uint8)                         # be 8-bit unsigned (np.uint8)
                                        # for GeoDict structures

  Structure =                           # reshape the 1-dimensional
voxel_values.reshape(nx, ny, nz)        # array voxel_values to a
                                        # 3D array of given dimensions
                                        # nx x ny x nz

  gd.setStructure(Structure, 1e-6)      # visualize the structure defined
                                        # in the csv file in GeoDict,
                                        # by passing the 3D numpy array
                                        # and assigning voxel length 1µm
```

gd.getStructureDescription

Returns a string, containing the structure description of the currently loaded structure. The description is to be found in the title bar of GeoDict or in the **Project Status Section** on the left, when the **Structure** settings are unfolded. It displays how the geometry was generated or saved.



Input: None

Example:

For an example, see underneath the **setStructureDescription** command.

gd.setStructureDescription

Sets the description text for the currently loaded structure.

Input:

- custom description, string

Example:

```
Struc_Des_old =                          # get current structure
gd.getStructureDescription()             description

gd.setStructureDescription("New          # changes description to "New
Description")                            Description"

Struc_Des_new =                          # gets new structure description
gd.getStructureDescription()

gd.msgBox(f'The structure                # outputs the description change
description was changed from
"{Struc_Des_old}" to
"{Struc_Des_new}".')
```

gd.getStructureHash

This function is deprecated, use gd.getStructureHash64 77 instead.

gd.getStructureHash64

Returns the new 64-bit structure hash (**Structure ID**) of the currently loaded structure as an integer. This can be used, e.g., to determine if a GDR result file corresponds to a given structure. This is a more robust unique identifier than **getStructureHash**.

Input: None

Example:

```
import stringmap                         # imports the Python module
                                         stringmap

gdr =                                    # assign the result file as a
stringmap.parseGDR('FiberGeo.gdr')       string
                                         # to the variable gdr

GDR_Hash_64 =                            # assign the ID of the structure
gdr['Geometry:Hash64']                   # corresponding to the result
                                         file
                                         # to the variable GDR_Hash
```

```
Structure_Hash_64 =                    # assign the ID of the loaded
gd.getStructureHash64()                structure
                                       # to the variable Structure_Hash

if int(GDR_Hash_64) ==                 # condition: if the IDs are equal,
Structure_Hash_64:                     the
                                       # following indented section is
                                       executed

  gd.msgBox("The loaded structure      # show message dialog
belongs to the result file
FiberGeo.gdr")

else:                                  # if the IDs are not equal, the
                                       following
                                       # indented section is executed

  gd.msgBox("The loaded structure      # show message dialog
does not belong to the result file
FiberGeo.gdr.")
```

gd.getStructureFileName

Returns the structure file name and relative file path of the currently loaded structure as a string.

Input: None

Example:

```
filename =                             # assigns the currently loaded
gd.getStructureFileName()              structure's
                                       # file name to the variable
                                       filename

gd.msgBox(f"Currently {filename}       # show message dialog
is loaded.")
```

gd.updateGeometry

This command has no return value but updates the geometry renderer.

Input: None

GAD-Object Functions

There are many possibilities to manipulate GAD-objects from Python. Here, API functions are given to get GAD-information. Use this GAD-information with the commands recordable from GadGeo to create special structures.

gd.getGADMode

Returns the GAD mode as an integer.

- 0: The current voxel geometry only consists of GAD-objects.
- 1: The current voxel geometry contains not only GAD-objects.
- 2: No GAD-objects are loaded.

Input: None

Example:

```
gad_mode = gd.getGADMode()              # assign GAD mode to variable
                                        # gad_mode

if gad_mode != 2:                       # condition: if the GAD mode
                                        # is not equal to 2,
                                        # i.e., 0 or 1,
                                        # the following indented
                                        # section is executed

  gd.msgBox(f"The structure             # show message dialog
contains GAD objects.")

else:                                   # if the condition above
                                        # is not true, i.e.,
                                        # the GAD mode is 2,
                                        # the following indented
                                        # section is executed

  gd.msgBox(f"The structure             # show message dialog
doesn't contain GAD objects.")
```

gd.getNumberOfGADObjects

Returns the number of loaded GAD objects as an integer.

Input: None

Example:

```
GAD_number =                            # get number of GAD objects
gd.getNumberOfGADObjects()

gd.msgBox(f"The structure contains      # show message dialog
{GAD_number} GAD objects.")
```

gd.getGADObject

Returns the settings of the GAD object with the given index id (first object has id 1) as a Python dictionary.

Input:

- GAD object ID, integer
- GeoDict version, string

Example:

For an example see getSelectedGADObjects 80 below.

gd.getSelectedGADObjects

Returns a list containing the IDs of the currently selected GAD objects.

Input: None

Example:

For the following example, a structure has to be loaded and one or more GAD Objects must be selected:

```
GAD_Selection =                          # get IDs of selected gad objects
gd.getSelectedGADObjects()

GAD_ID = GAD_Selection[0]                # choose smallest selected GAD
                                         object ID

GAD_Object =                             # get the settings of the
gd.getGADObject(GAD_ID," 2024")          corresponding gad object

gd.msgBox(GAD_Object['Type'])            # show type of selected GAD_object
                                         in message
                                         # box, e.g., sphere, ellipsoid,
                                         circular fiber, …
```

Result Functions

Get information about the currently loaded volume files. Manipulate volume files by getting the data as a 3D numpy array, editing it and updating the volume field in the end.

gd.getResultFile

Returns a Python dictionary containing the parameters of the given GeoDict result file (*.gdr). The result file must be loaded in the Result Viewer. Use for example the gd.showGDR 96 function for loading result files. While this function directly returns a Python dictionary, instead of a stringmap object, no units are contained in the dictionary. If units are relevant, use the stringmap 137 module instead.

Input:

- result file ID as integer **OR** file name as string
  - for the ID the loaded result files are numbered top to bottom (counting starts with 0)

Example:

```
gdr =                                   # get result file and
gd.getResultFile('FiberGeo.gdr')        # assign it to variable
                                        # gdr

svp_reached = gdr['ResultMap']          # get reached solid volume
['SolidVolumePercentage']               # percentage from result map and
                                        # assign it to variable
                                        svp_reached

svp_target  = gdr['InputMap']           # get target solid volume
['SolidVolumePercentage']               # percentage from input map and
                                        # assign it to variable svp_target

svp_reached,svp_target =                # transform string values to float
float(svp_reached),float(svp_targe
t)

svp_error   = abs(svp_reached -         # compute relative error
svp_target)/svp_target                  # between reached and
                                        # target svp

print(svp_error)                        # print the computed error
```

gd.getVolumeFieldsInfo

Returns a list of dictionaries describing the currently loaded Volume Fields (Result fields, e.g., Flow results). The "index" field of each entry gives the index to use for the following function.

Input: None

Example:

For an example, see below under the **getVolumeField** 81 command.

gd.getVolumeField

This function returns a numpy array for a currently loaded Volume Field. There are separate Fields for each component, e.g., for a flow field there are separate fields for VelocityX, VelocityY, VelocityZ and Pressure. If the needed index or name is unknown, the previous function **gd.getVolumeFieldsInfo** 81 can be used to obtain the desired information.

Input:

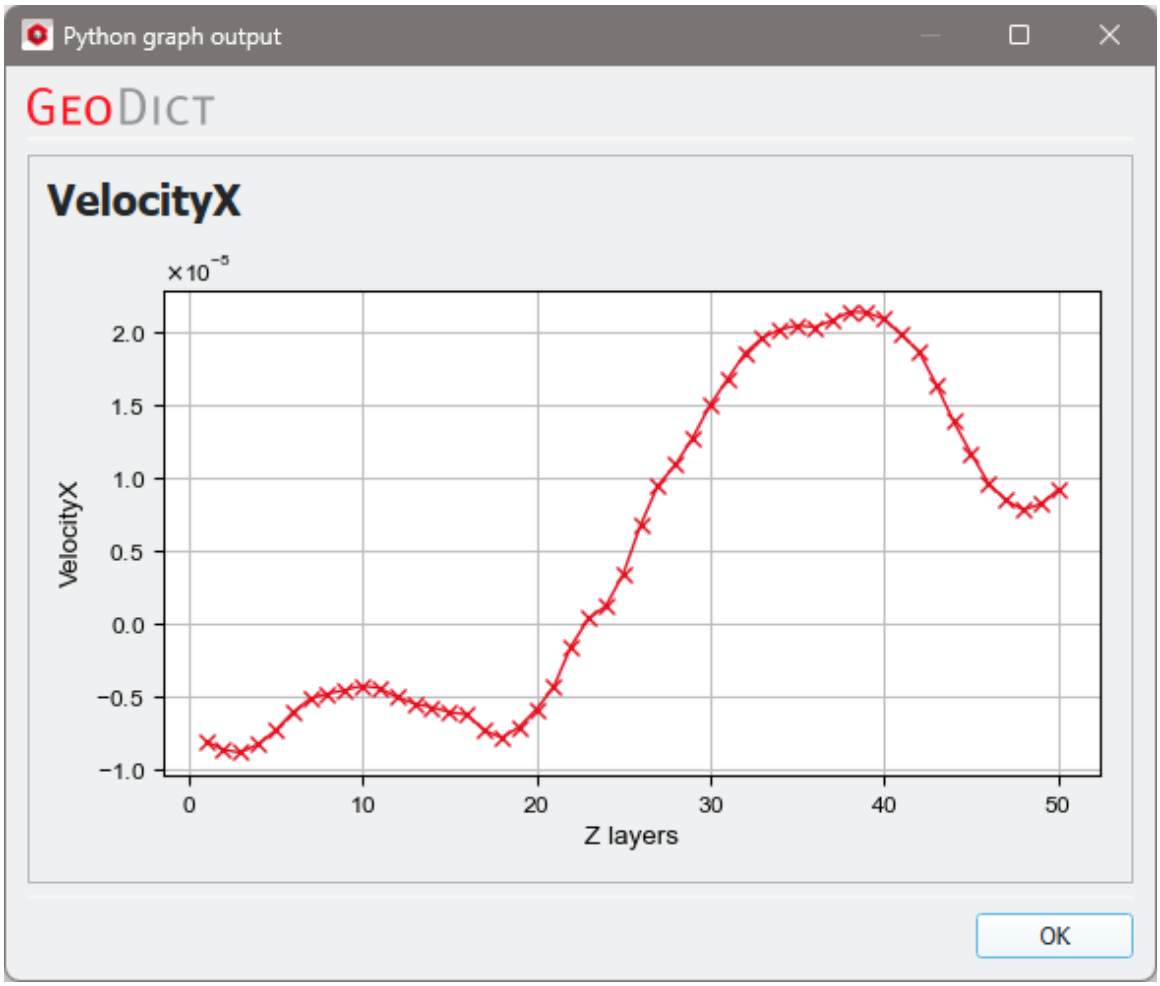
- Volume Field index as integer **OR** Volume Field name as string

Example:

For example, this function can be used to compute statistics from the results. In the following example for the first of the loaded volume fields a statistic over the Z-layers is plotted in a graph, using another GeoDict API function. A volume field must be loaded and, if the volume field is a simulation result, also the corresponding structure.

```
VolumeInfo =                          # get list of dictionaries of
gd.getVolumeFieldsInfo()              loaded
                                      # Volume Fields and assign it to
                                      # variable VolumeInfo

print(VolumeInfo)                     # print all data contained in
                                      VolumeInfo
                                      # to console / logfile

nx, ny, nz = gd.getVolDimensions()    # get the number of voxels in
                                      # X-, Y- and Z-direction and
                                      # assign the numbers to variables

Structure = gd.getStructure()         # assign 3D numpy array describing
                                      # the loaded structure to the
                                      # variable Structure

Name = VolumeInfo[0]['name']          # assign the name of the first
                                      # volume field to
                                      # the variable Name

VolumeField =                         # assign the numpy array
gd.getVolumeField(Name)               describing
                                      # the volume field
                                      # to the variable VolumeField

Statistic = []                        # Create empty list to store
                                      # the statistical values

for k in range(nz):                   # loop over all Z-layers

  value_sum = 0                       # creating variable value_sum to
                                      # sum up the result values

  value_count = 0                     # creating variable value_count
                                      # to count the summands

  for j in range(ny):                 # loop over all Y-layers

    for i in range(nx):               # loop over all X-layers

      if Structure[i][j][k] == 0:     # condition: if the kth Z-value
                                      in
                                      # the jth Y-column of the ith X-
                                      layer
                                      # is pore space (ID 0), the
                                      following
                                      # indented section is executed

        value_sum = value_sum +       # add all pore space result
VolumeField[i][j][k]                  values
                                      # of the kth Z-layer to the sum
                                      value_sum

        value_count = value_count     # count the summands of value_sum
+ 1
```

```
  meanVal = value_sum /                 # compute mean value of all pore
value_count                             # space result values in the kth
                                        # Z-layer and assign it to the
                                        # variable meanVal

  Statistic.append(meanVal)             # append the mean value of the
                                        # Z-layer to the Statistic list

gDlg = gd.makeGraphDialog()             # create a graph dialog object

graph = gDlg.addGraph(Name, "Z          # add a graph object with the
layers", Name)                          name
                                        # of the volume field as title
                                        and
                                        # Y-axis legend and Z-layers as
                                        # X-axis legend

Z_layers = list(range(1, nz + 1))       # writes the Z-layer numbers
                                        # 1, 2, …, nz-1, nz into a
                                        # list named Z-layers

graph.addData(Z_layers, Statistic,      # add a single dataset with the
Name)                                   # Z-layers as X-values, the mean
                                        # result values as Y-values and
                                        # the name of the volume field
                                        # as legend to this graph

gDlg.run()                              # show graph dialog
```



gd.updateVolumeField

This command has no return value but updates the visualization of a volume field, for example after changing values in the numpy array obtained with gd.getVolumeField 81 .

Input:

- Volume Field index as integer **OR** Volume Field name as string

gd.getNumberOfTriangles

Returns number of triangles in the current surface mesh. If no mesh is loaded 0 is returned.

Input: None

Example:

```
Number = gd.getNumberOfTriangles()  # assigns the number of triangles
                                    # to the variable Number
```

```
gd.msgBox ( f"The number of triangles is {Number}")          # show message dialog
```

gd.getTriangulationBoundingBox

Returns the bounding box of the current triangulation. If no triangulation exists {{0,0,0}, {0,0,0}} is returned.

Input: None

Example:

```
Box = gd.getTriangulationBoundingBox()          # assigns the host name
                                                # to the variable Host_name

X = Box[1][0]*10**6                             # get the first entry of the
                                                # second dictionary, i.e.,
                                                # the X-dimension in m,
                                                # transform it to µm and assign
                                                # it to the variable X

gd.msgBox ( f"The bounding box has {X} µm in X-direction.")   # show the result in
                                                              # a message dialog
```

Material Functions

Get information about the constituent materials and the GeoDict material database.

gd.getConstituentMaterials

Returns the map of the current constituent materials as Python dictionary.

Input: None

Example:

```
Materials = gd.getConstituentMaterials()        # get dictionary of constituent
                                                materials
                                                # and assign it to variable
                                                Materials

ID_0_Type = Materials['Material00']['Type']     # get type of material ID 0 and
                                                assign
                                                # it to variable ID_0_Type

gd.msgBox(f"Material ID 00 is of type {ID_0_Type}.")   # show message dialog of result
```

gd.getDataBaseMaterial

Returns the information of the given material in the GeoDict material database as Python dictionary.

Input:

- material name, string

Example:

```
Material_Air =                          # get the data base
gd.getDataBaseMaterial("Air")           # information for air

air_dens    = Material_Air["Flow"]      # get the sixth entry
["MaterialLaw1"]["Density"][0][6]       # in the density list
                                        # for air
                                        # counting starts with 0

air_dens_u = Material_Air["Flow"]       # get the unit for
["MaterialLaw1"]["Density"][1]          # the density

air_temp    = Material_Air["Flow"]      # get the sixth entry
["MaterialLaw1"]["Temperature"][0]      # in the temperature
[6]                                     # list for air
                                        # counting starts with 0

air_temp_u = Material_Air["Flow"]       # get the unit for
["MaterialLaw1"]["Temperature"][1]      # the temperature

gd.msgBox(f"At {air_temp} degrees       # show message dialog
{air_temp_u} the density of air is
{air_dens} {air_dens_u}.")
```

gd.getMaterialDataBaseFolder

Returns the folder path of the material data base folder containing the defined materials and their parameters as a string.

Input: None

Example:

```
databasefolder =                        # get the data base folder path
gd.getMaterialDataBaseFolder()

import os                               # import the Python module os,
                                        which has many
                                        # useful functions regarding file
                                        directories

foldercontent =                         # get the content of the folder
os.listdir(databasefolder)              # as a list of strings

print(foldercontent)                    # print the list to the GeoDict
                                        console,
                                        # resulting in a list of *.txt
                                        files
                                        # corresponding to the contained
                                        materials
```

gd.setTemperature

Sets the current temperature for the simulation solvers to change the constituent material properties accordingly. The available units are Celsius, Kelvin and Fahrenheit.

Input:

- temperature, float
- unit, string

Example:

```
gd.setTemperature(15,"Celsius")      # set the temperature to 15 °C
```

GeoDict Settings Functions

Get information about GeoDict settings, as release version or location of different folders.

gd.getSettingsFolder

Returns the settings folder as a string.

Windows: c:\user\%USERNAME%\GeoDict2026

Linux: ~/.geodict2026

Input: None

Example:

```
SettingsFolder =                     # assigns the file path of
gd.getSettingsFolder()               # the settings folder to the
                                     # variable SettingsFolder

gd.msgBox(f"The GeoDict settings     # shows the settings folder
can be found in                      # in a message box
\n{SettingsFolder}")
```

gd.getInstallationFolder

Returns the directory that contains the GeoDict executable as a string.

Input: None

Example:

```
InstallationFolder =                 # assigns the file path of the
gd.getInstallationFolder()           # installation folder to the
                                     # variable InstallationFolder

gd.msgBox(f"The GeoDict executable   # shows the installation folder
is found in \n                       # in a message box
{InstallationFolder}")
```

gd.getProjectFolder

Returns the current project folder of GeoDict as a string.

Input: None

Example:

```
projectfolder =                          # assigns the file path of the
gd.getProjectFolder()                    current
                                         # project folder to the variable
                                         projectfolder

gd.showGDR (projectfolder +              # opens the GeoDict result file
"/example.gdr")                          # "example.gdr" located in the
                                         # current project folder
```

gd.getHostName

Returns the name of the host as a string.

Input: None

Example:

```
Host_name = gd.getHostName()             # assigns the host name
                                         # to the variable Host_name

gd.msgBox ( f"The host is                # show message dialog
{Host_name}")
```

gd.getStandardFileHeader

Returns the Python dictionary for the standard header that is used in recorded macros as a string.

Input: None

Example:

```
Header =                                 # assigns the string of the
gd.getStandardFileHeader()               standard
                                         # file header to the variable
                                         Header

gd.msgBox(Header)                        # show the standard file header
                                         # in a message dialog
```

gd.getVersion

Returns the current GeoDict version as a string.

Input: None

Example:

```
Version = gd.getVersion()              # assigns the version as a
                                       # string to the variable Version

gd.msgBox ( f"The current GeoDict      # show message dialog
version is {Version}")
```

gd.getVersionInfo

Returns the Python dictionary for the standard header that is used in recorded macros, containing the GeoDict version, revision and release date.

Input: None

Example:

```
Header = gd.getVersionInfo()           # assigns the standard
                                       # file header to the
                                       # variable Header

gd.msgBox (f"The current GeoDict       # show the current
revision is {Header['Revision']}")     # revision in a
                                       # message dialog
```

gd.get3rdPartyBinFolder / gd.getResourcesFolder / gd.getGDModulesFolder

Gets the directory that contains the 3rd party binaries / GeoDict resources / GeoDict module folders as a string. Usually, for non-developers all three data groups are contained in the GeoDict installation folder. Thus, use getInstallationFolder 87 instead.

Input: None

gd.getGDFolder

Gets the directory that contains the GeoPy API files as a string.

Input: None

gd.getGDGeoAppsFolder

Gets the directory that contains the built-in GeoDict GeoApps as a string.

Input: None

gd.getGDVideoMacroFolder

Gets the directory that contains the built-in video macros as a string.

Input: None

Macro Functions

Get information about the current macro.

gd.getMacroFileFolder

Returns the directory that contains the macro file as a string.

Input: None

Example:

```
macrofolder =                          # assigns the file path of the
gd.getMacroFileFolder()                macro
                                       # to the variable macrofolder

gd.showGDR (macrofolder +              # opens the GeoDict result file
"/example.gdr")                        # "example.gdr" located in the
                                       # same folder as the macro.
```

gd.getMacroFileName

Returns the macro name as a string.

Input: None

Example:

```
macroname = gd.getMacroFileName()      # assigns the macro name
                                       # to the variable macroname

gd.msgBox (f"{macroname} is            # shows the macrofilename
running.")                             # in a message box.
```

gd.getCurrentVariableMap

Returns a Python dictionary consisting of the variable names as keys and their corresponding values.

Input: None

Example:

```
VarMap =                               # get variable map and
gd.getCurrentVariableMap()             # assign it to variable
                                       # VarMap

print(VarMap)                          # print variable map
                                       # to console
```

gd.setCurrentVariableMap

After inputting a Python dictionary consisting of the variable names as keys and their corresponding values, this command sets a new variable map internally changing the output of the function gd.getVariableMap.

Input:

- Variable map, Python dictionary

Example:

```
gd.setCurrentVariableMap({'gd_SVP'  # set new variable map
:(15.5,'%'),'gd_RS':1})

VarMap =                            # get variable map and
gd.getCurrentVariableMap()          # assign it to variable
                                    # VarMap

print(VarMap)                       # print variable map
                                    # to console
```

Progress Bar Functions

Set-up a progress bar for your macro. Using a progress bar helps you and other users of your macro to estimate the remaining time for the macro execution.

gd.getProgress

This command has no return value but creates a progress bar object that is shown in GeoDict with the passed number of steps and the passed text as description. The progress bar remains visible until the object runs out of scope or is explicitly deleted. It is possible to use the progress bar as a context manager.

Input:

- progress bar name, string
- total number of steps, integer
- splash screen name, string (optional)
  - Displays the given splash screen in the progress dialog. Entering a random string displays the default GeoDict splash screen. Omit parameter or enter an empty string (") to obtain a progress dialog without a splash screen.
- show graph, bool (optional)
  - Add a graph to progress dialog if True is entered. No graph is displayed if the parameter is omitted or set to False.
- show stop button (optional)
  - Add a stop button to the progress dialog if set to True. No stop button will be added if parameter is omitted or set to False.

Sub functions:

.update

Updates the progress bar to the specified step. I

Input:

- Input: step number, integer

.updateWithGraph

Updates the progress bar to the specified step and also displays and updates a graph with the given values.

Input:

- step number, integer
- X-axis label, string
- X-value, integer or float
- Y-axis label, string
- Y-value, integer or float

.wasCancelled

Checks if the cancel button was hit and returns True or False.

Input: None

.wasStopped

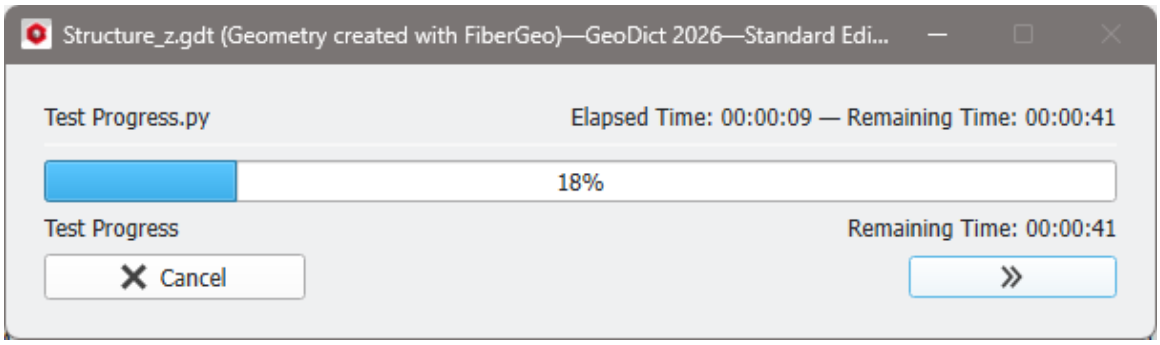
Checks if the stop button was hit and returns True or False.

Input: None

Example 1 - simple

```
from time import sleep          # import the Python module "sleep"
                                # we use this to artificially prolong
                                # the macro execution in this example

progress =                      # create a progress bar about 100 steps
gd.getProgress("Test            # that is named "Test Progress"
Progress", 100)

for i in range(101):            # start a loop doing the same tasks
```

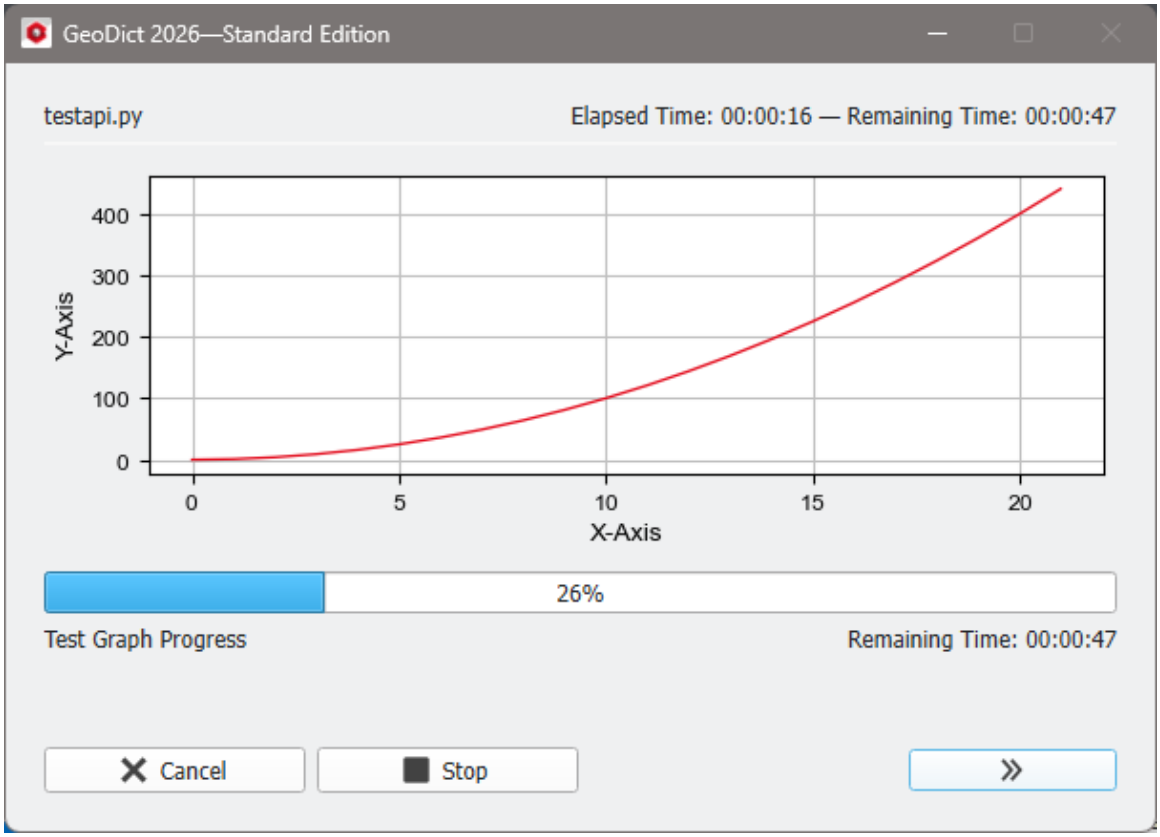
```
                              # for i = 0, …, 100

  sleep(0.5)                  # wait for 0.5 seconds
                              # prolongs macro execution to have
                              # an example progress bar

  progress.update(i)          # update the progress bar to step i

  if                          # condition that if the Cancel button
progress.wasCancelled():      # was hit, the loop is stopped
    break

del progress                  # delete the progress bar
```



Example 2 - with graph

```
from time import sleep        # import the Python module "sleep"
                              # we use this to artificially prolong
                              # the macro execution in this example

progress2 =                   # create a second progress
gd.getProgress("Test Graph    # bar about 80 steps
Progress", 80, '', True,      # that is named
True)                         # "Test Graph Progress".
                              # A graph and a Stop button
                              # will be added
                              # to the progress dialog

for i in range(81):           # start a loop doing
                              # the same tasks for
                              # i = 0, …, 80

  x = i                       # set X-value for graph
                              # to iteration value

  y = x**2                    # set Y-value for
                              # graph to x²

  sleep(0.5)                  # wait for 0.5 seconds
                              # prolongs macro execution to have
                              # an example progress bar
```

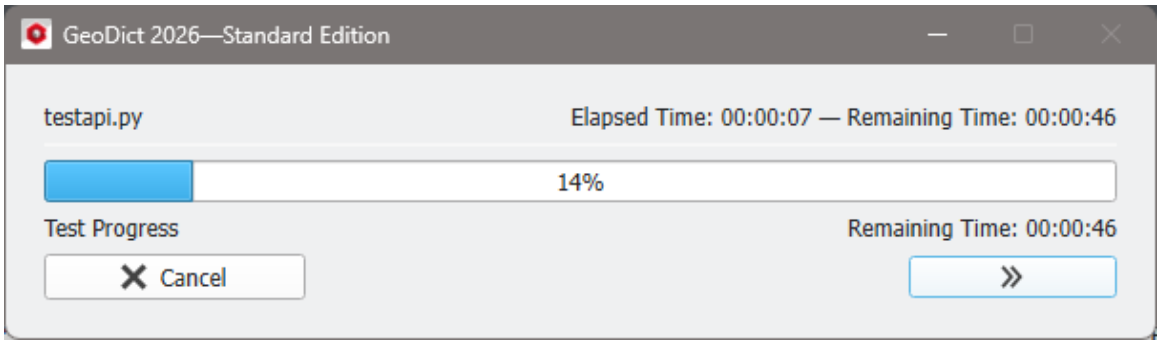
```
  progress2.updateWithGrap  # update the progress bar
h(i, "X-Axis", x, 'Y-       # to step i and the
Axis', y)                   # graph with the
                            # value pair (x,y)

  if                        # condition that if the
progress2.wasStopped():     # Stop button was hit,
     break                  # the loop is stopped

del progress2               # delete the progress bar
```



Example 3 - Use "with" instead of "del"

```
from time import sleep      # import the Python module "sleep"
                            # we use this to artificially prolong
                            # the macro execution in this example

with gd.getProgress("Test   # create a progress bar about 100 steps
Progress", 100) as          # that is named "Test Progress"
progress3:                  # progress disappears after the last line
                            # of the following indented block

  for i in range(101):      # start a loop doing the same tasks
                            # for i = 0, …, 100

    sleep(0.5)              # wait for 0.5 seconds
```

```
                                # prolongs macro execution to have
                                # an example progress bar

    progress3.update(i)         # update the progress bar to step i

    if                          # condition that if the Cancel button
progress3.wasCancelled():       # was hit, the loop is stopped
        break
```



Pop-Up Functions

Sometimes you don't want pop-up dialogs to show and sometimes you want to show some custom messages. Use the following API functions to do so.

gd.getBlocker

Get a Blocker object to prevent GeoDict Dialogs from showing up. Use this function via 'with' keyword.

Input: None

Example:

For example, if saving images from a macro from different structures with a solid ID not shown, GeoDict will ask for every structure loading, if this material ID should be visualized. But the blocker command prevents the dialogs from showing up:

```
with gd.getBlocker():                   # blocks GeoDict dialogs
                                        # while the following
                                        # indented section
                                        # is executed

  for i in range(3):                    # loops over the indices
                                        # 0,1,2

    gd.runCmd("GeoDict:LoadFile",       # loads a GeoDict
{'FileName' : f'example{i}.gdt'},       # structure file
'2026')

    SaveThreeDImage_args = {            # GeoPy dictionary
                                        # containing the
                                        # parameters
```

```
                                    # to save an image

      'FileName' :
f'example{i}.png',

      'Resolution': {'Mode' :
'Current'},

      'IncludeTransparency' :
False}

    gd.runCmd("GeoDict:Save3DImage  # GeoDict command to
", SaveThreeDImage_args, '2026')    # save an image
```

gd.msgBox

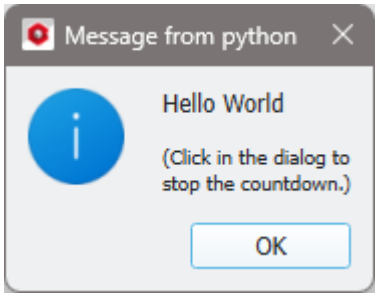
Displays a simple message box containing the given basic Python value (string, integer, float, ...) and an OK button. The execution continues after clicking OK. This function is useful for debugging. The command has no return value.

Input:

- message, any basic Python value (integer, float, string, bool, ...)

Example:

```
gd.msgBox("Hello World")            # show message dialog
```



gd.showGDR

This will open the given GDR file contents within a GeoDict dialog. The command has no return value.

Input:

- gdr file path, string

Example:

For example, if a result file with the name Example.gdr is saved, it can be opened in the **Result Viewer** with this command:

```
gd.showGDR("Example.gdr")           # opens the file in the Result
                                    Viewer
```

Dialog Class

Create your own customized dialog for variable input or messages.

gd.makeDialog

Create an input dialog object to query the user for parameters. There is a large variety of possible input fields.

Input:

- key name, string
  - This is used to store dialog settings in the settings map. Use a unique key for each dialog unless you are re-using the same dialog and want their settings to affect each other.
- title, string (optional)
  - The title is shown as the window title of the dialog. If no title is given, geodict2026 is shown.

Sub functions:

Note! In the examples for the input parameters often the notation "parameter name = parameter value" is used. This is not mandatory, but using this notation, the order and the number of optional arguments used does not matter anymore. This is very useful, if you only want to enter some of the optional parameters. Additionally it increases readability, highlighting which value belongs to which key. This can sometimes also be useful for obligatory arguments.

.addBoolInput

Add a check box to your dialog. The value is True, when the checkbox is enabled, otherwise False

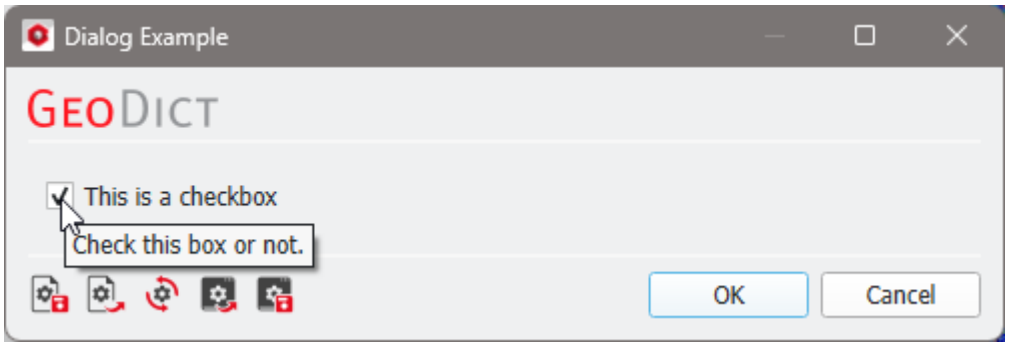
Input:

- key, string
- description, string
- init, bool (optional)
- tooltip, string (optional)

Example:

```
dlg = gd.makeDialog("MyDialog",      # create a dialog object
"Dialog Example")                    # with a key and a title
```

```
dlg.addBoolInput("myBooleanParamet  # create a check box,
er",                                # with key, description,
  "This is a checkbox",             # default value and tooltip
init=True,
  tooltip="Check this box or
not.")

result = dlg.run()                  # open the dialog with all the
                                    # settings assigned to the dialog
                                    # variable, here dlg
```



.addIntegerInput

Add an integer input to your dialog. Here, integer numbers can be entered in the corresponding field.

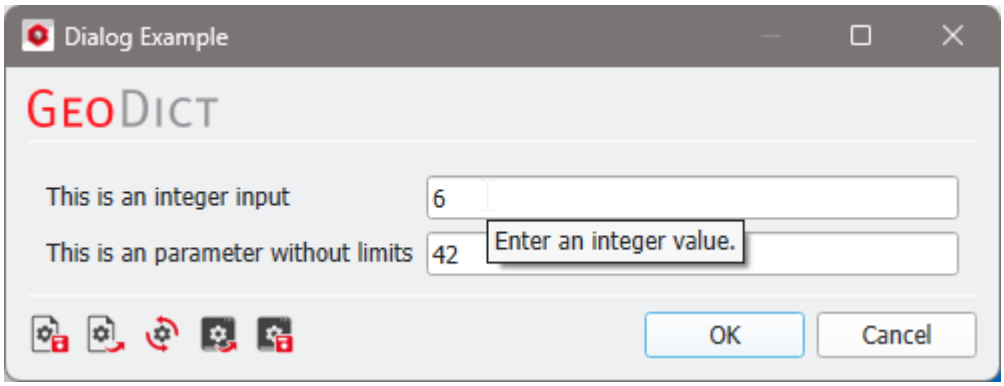
Input:

- key, string
- description, string
- min, integer (optional)
- max, integer (optional)
- init, integer (optional)
- tooltip, string (optional)

Example:

```
dlg = gd.makeDialog("MyDialog",     # create a dialog object
"Dialog Example")                   # with a key and a title

dlg.addIntegerInput("myIntegerPara  # create an input field
meter",                             # for integer values with key,
  "This is an integer input",       # description, boundaries,
min=5, max=10,                      # default value and tooltip
  init=6, tooltip="Enter an
integer value.")
```

```
dlg.addIntegerInput("myNewIntegerP  # create an input field
arameter",                          # for integer values with key,
  "This is an parameter without     # description, default value
limits",                            # and tooltip
  init=42, tooltip="Enter some
value here.")

result = dlg.run()                  # open the dialog with all the
                                    # settings assigned to the dialog
                                    # variable, here dlg
```



.addUintegerInput

Add an uinteger input field to your dialog. Here, the value can also be changed by clicking the arrows on the right.

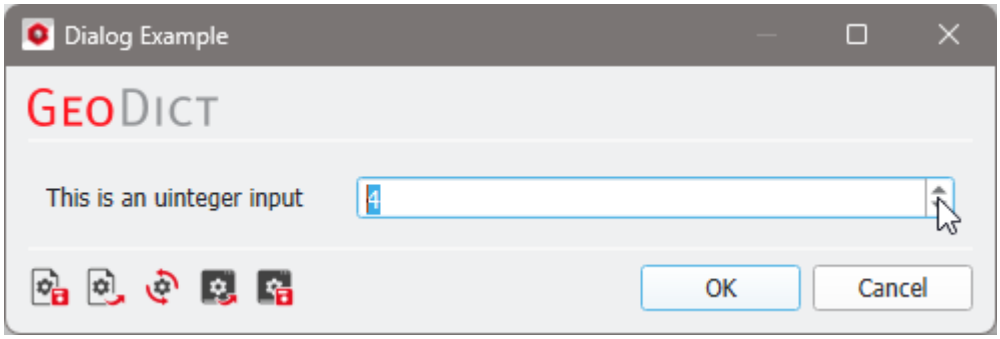
Input:

- key, string
- description, string
- min, integer (optional)
- max, integer (optional)
- init, integer (optional)
- tooltip, string (optional)

Example:

```
dlg = gd.makeDialog("MyDialog",     # create a dialog object
"Dialog Example")                   # with a key and a title

dlg.addUintegerInput("MyUintInput"  # create an input field
,                                   # for uinteger values with key,
  "This is an uinteger input",      # description, boundaries,
  min = -5, max = 5, init=0,        # default value and tooltip
  tooltip="Choose an integer
parameter")
```

```
result = dlg.run()                    # open the dialog with all the
                                      # settings assigned to the dialog
                                      # variable, here dlg
```



.addFloatInput

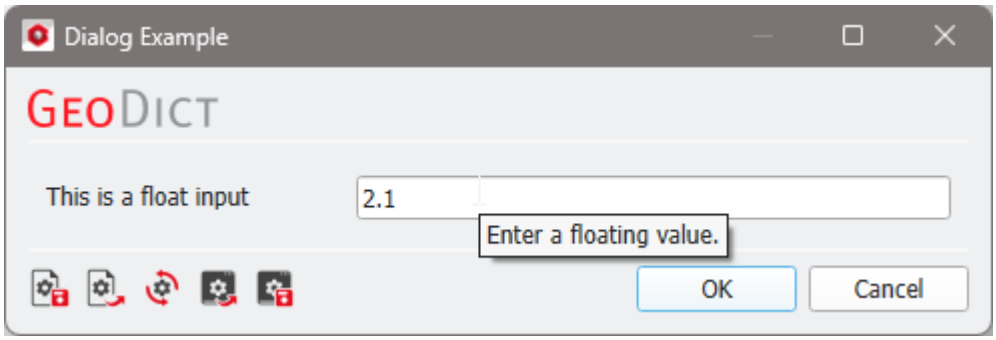
Add an floating point number input field. Here, floating point numbers can be entered.

Input:

- key, string
- description, string
- min, float (optional)
- max, float (optional)
- init, float (optional)
- tooltip, string (optional)

Example:

```
dlg = gd.makeDialog("MyDialog",       # create a dialog object
"Dialog Example")                     # with a key and a title

dlg.addFloatInput("myFloatParamete    # create an input field
r",                                   # for float values with key,
  "This is a float input",            # description, boundaries,
  min = -3.5, max = 5.2, init=2.1,    # default value and tooltip

  tooltip="Enter a floating
value.")

result = dlg.run()                    # open the dialog with all the
                                      # settings assigned to the dialog
                                      # variable, here dlg
```



.addTextInput

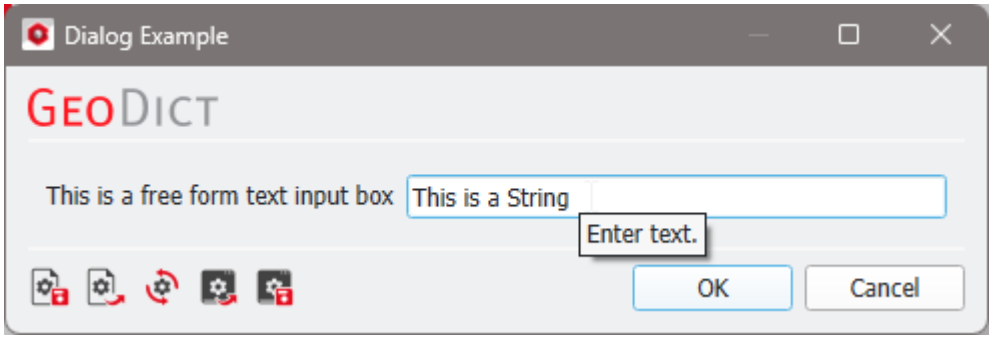
Add an text input field to your dialog. Here, text can be entered and will be considered as string.

Input:

- key, string
- description, string
- init, string (optional)
- tooltip, string (optional)

Example:

```
dlg = gd.makeDialog("MyDialog",          # create a dialog object
"Dialog Example")                        # with a key and a title

dlg.addTextInput("myStringParamete       # create an input field
r",                                      # for text with key,
  "This is a free form text input        # description, default value
box",                                    # and tooltip
  init="This is a String",
  tooltip="Enter text.")

result = dlg.run()                       # open the dialog with all the
                                         # settings assigned to the dialog
                                         # variable, here dlg
```



.addFileInput

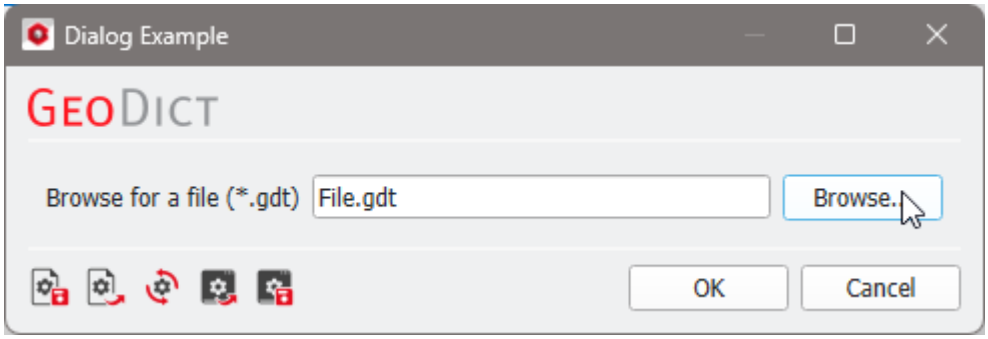
Add a file input field. The input box comes with a Browse button to browse for the file. The value is considered as string.

Input:

- key, string
- description, string
- file type, string (optional)
- init, string (optional)
- tooltip, string (optional)

Example:

```
dlg = gd.makeDialog("MyDialog",      # create a dialog object
"Dialog Example")                    # with a key and a title

dlg.addFileInput("myFileSelection"   # create an input field for
,                                    # gdt files with key,
  "Browse for a file", "gdt",        # description, default value
  init="File.gdt",                   # and tooltip
  tooltip="Browse for a gdt
file.")

result = dlg.run()                   # open the dialog with all the
                                     # settings assigned to the dialog
                                     # variable, here dlg
```



.addFolderInput

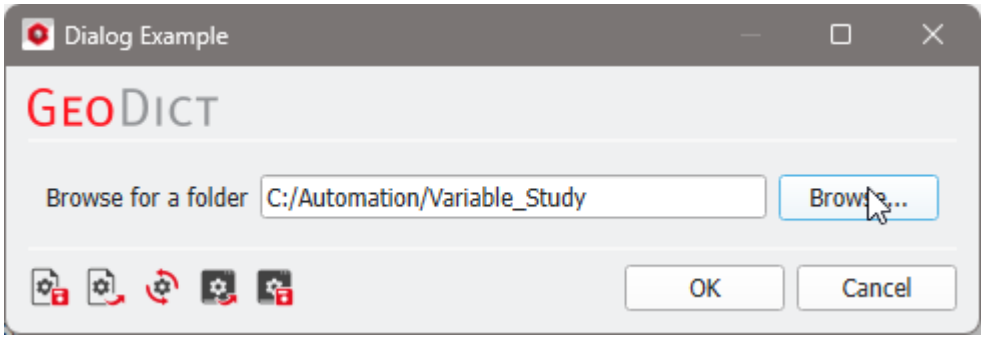
Add a folder input field. The input box comes with a Browse button to browse for the folder. The value is considered as string.

Input:

- key, string
- description, string
- init, string (optional)
- tooltip, string (optional)

Example:

```
dlg = gd.makeDialog("MyDialog", "Di # create a dialog object
                                    # with a key and a title

dlg.addFolderInput("MyFolderInput", # create an input field for
  "Browse for a folder")            # folder paths with key,
                                    # and description

result = dlg.run()                  # open the dialog with all the
                                    # settings assigned to the dialog
                                    # variable, here dlg
```



.addComboInput

Add a combo box to your dialog. The value choice is given as list of strings. Within the script, the returned value is the index of the selected list item.

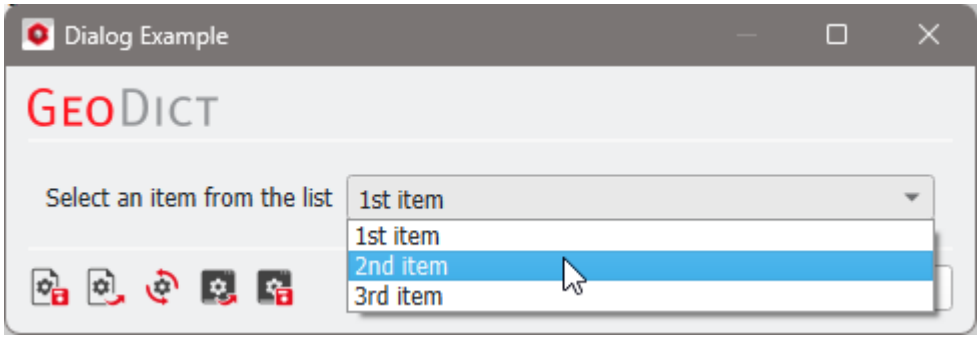
Input:

- key, string
- description, string
- selection, list of strings
- tooltip, string (optional)

Example:

```
dlg = gd.makeDialog("MyDialog",     # create a dialog object
"Dialog Example")                   # with a key and a title

dlg.addComboInput("myComboBox",     # create a pull-down menu
  "Select an item from the list",   # with the given values for
  ["1st item", "2nd item", "3rd     # selection.
item"])                             # Key and description are given.
                                    # The returned value is the
                                    # index of the selected item,
                                    # e.g., 0 for the first item,
                                    # 1 for the second etc.

result = dlg.run()                  # open the dialog with all the
                                    # settings assigned to the dialog
```

```
                                    # variable, here dlg
```



.addComboInputString

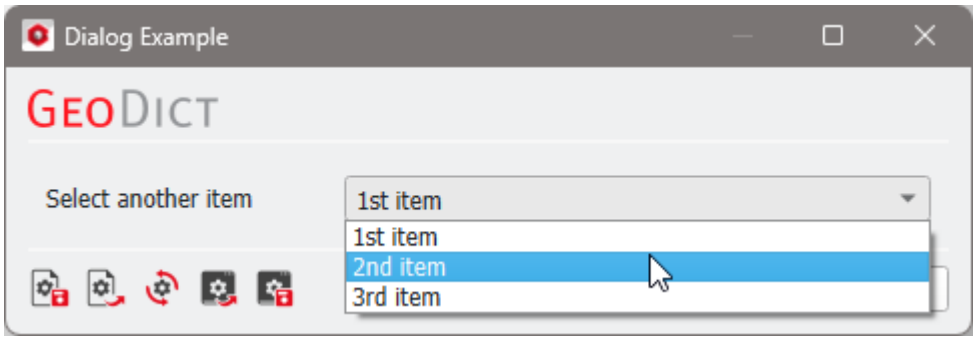
Add a combo box to your dialog. The value choice is given as list of strings. Within the script, the returned value is the string of the selected list item.

Input:

- key, string
- description, string
- selection, list of strings
- tooltip, string (optional)

Example:

```
dlg = gd.makeDialog("MyDialog",          # create a dialog object
"Dialog Example")                        # with a key and a title

dlg.addComboInputString("ComboStri       # create a pull-down menu
ng",                                     # with the given values for
  "Select another item",                 # selection. Key and description
  ["1st item", "2nd item", "3rd          # are given.
item"])                                  # The returned value is
                                         # the string of the
                                         # selected item

result = dlg.run()                       # open the dialog with all the
                                         # settings assigned to the dialog
                                         # variable, here dlg
```



.addMaterialInput

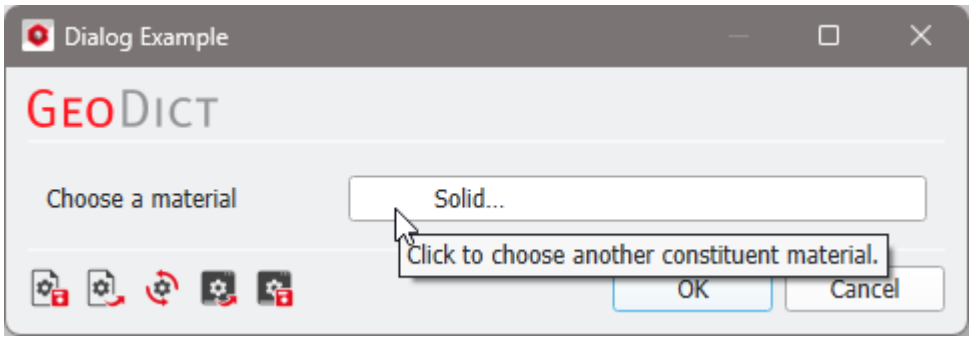
Add a material input box to your dialog. A material can be chosen from the GeoDict material database.

Input:

- key, string
- description, string

Example:

```
dlg = gd.makeDialog("MyDialog",          # create a dialog object
"Dialog Example")                        # with a key and a title

dlg.addMaterialInput("MyMaterialIn       # create an input button for
put",                                    # browsing the material data
  "Choose a material")                   # base. Key and description are
                                         given.

result = dlg.run()                       # open the dialog with all the
                                         # settings assigned to the dialog
                                         # variable, here dlg
```



.addTableInput

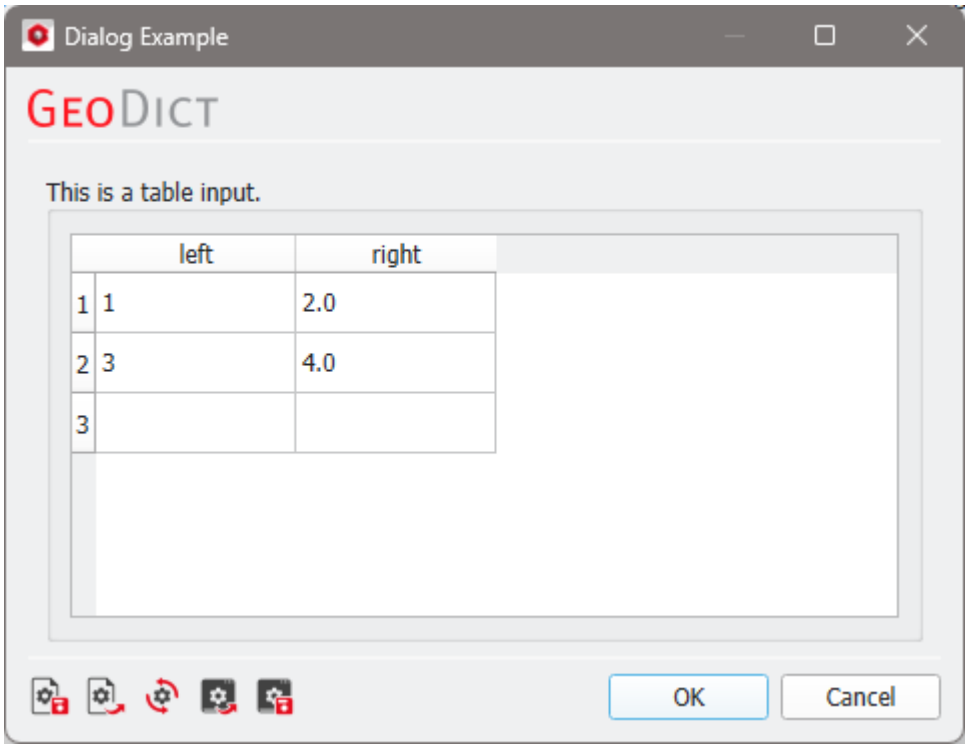
Add table input to your dialog. A table with multiple rows and the defined number of columns is given.

Input:

- key, string
- description, string
- types, list of strings
- columnHeaders, list of strings
- init, list of given types (optional)
- tooltip, string (optional)

Example:

```
dlg = gd.makeDialog("MyDialog",          # create a dialog object
"Dialog Example")                        # with a key and a title

dlg.addTableInput("MyTableInput",        # create a table with two columns,
  "This is a table input.",              # one for integer input
  types = "int,float",                   # and one for float.
  columnHeaders=["left","right"],        # Two default rows are given.,
  init=[[1,2.0],[3,4.0]])                # additionally to the mandatory
                                         # key, description, types and
                                         headers

result = dlg.run()                       # open the dialog with all the
                                         # settings assigned to the dialog
                                         # variable, here dlg
```



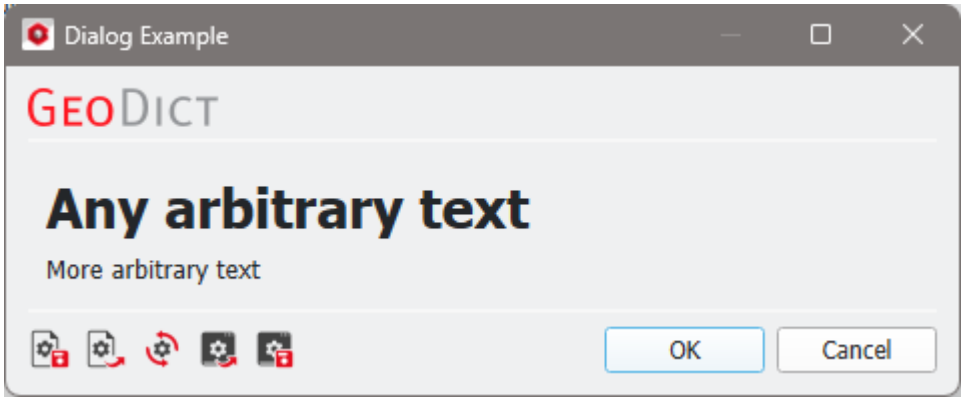
.addText

Add free-form text to your dialog in the given font size and in bold (True) or not bold (False).

Input:

- key, string
- description, string
- font size, integer (optional, default is 8))
- setBold, bool (optional, default is false)

```
dlg = gd.makeDialog("MyDialog",      # create a dialog object
"Dialog Example")                    # with a key and a title

dlg.addText("Any arbitrary text",    # add bold text to the
  20, True)                          # dialog with font size 20

dlg.addText("More arbitrary text",   # add text to the dialog
                                     # with font size 10
  10, False)                         # and not bold

result = dlg.run()                   # open the dialog with all the
                                     # settings assigned to the dialog
                                     # variable, here dlg
```



.beginGroup

Fields can be grouped within a box. Add input values or free text between the beginGroup and endGroup lines.

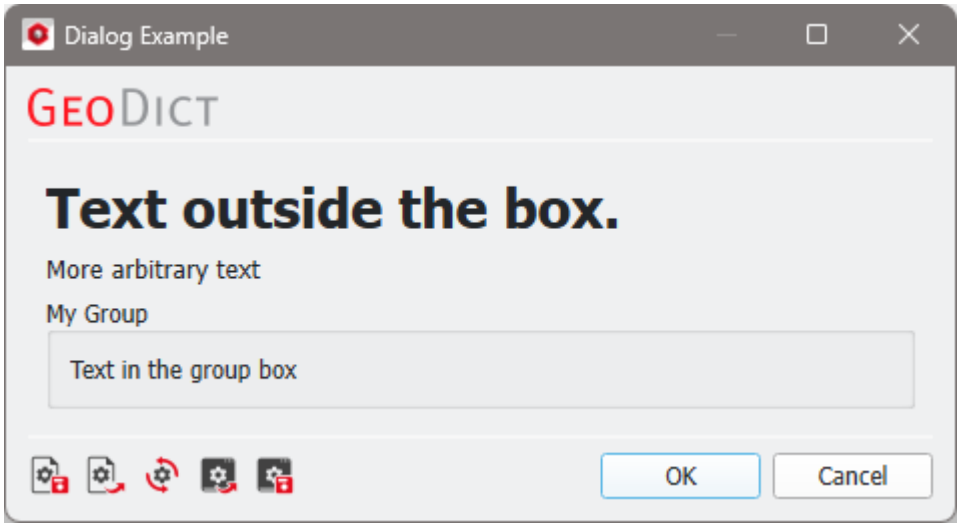
Input:

- group name, string

Example:

```
dlg = gd.makeDialog("MyDialog",      # create a dialog object
"Dialog Example")                    # with a key and a title
```

```
dlg.addText("Text outside the box.",
  20, True)                          # add bold text to the
                                     # dialog with font size 20

dlg.addText("More arbitrary text",   # add text to the dialog
                                     # with font size 10
  10, False)                         # and not bold

dlg.beginGroup("My Group")           # start a group
                                     # to add a box

dlg.addText("Text in the group box") # add any content inside
                                     # box, here we added
                                     # only some text

dlg.endGroup()                       # end the group

result = dlg.run()                   # open the dialog with all the
                                     # settings assigned to the dialog
                                     # variable, here dlg
```



.endGroup

Fields can be grouped within a box. Add input values or free text between the beginGroup and endGroup lines.

Input: None

Example: see beginGroup 107.

.addImage

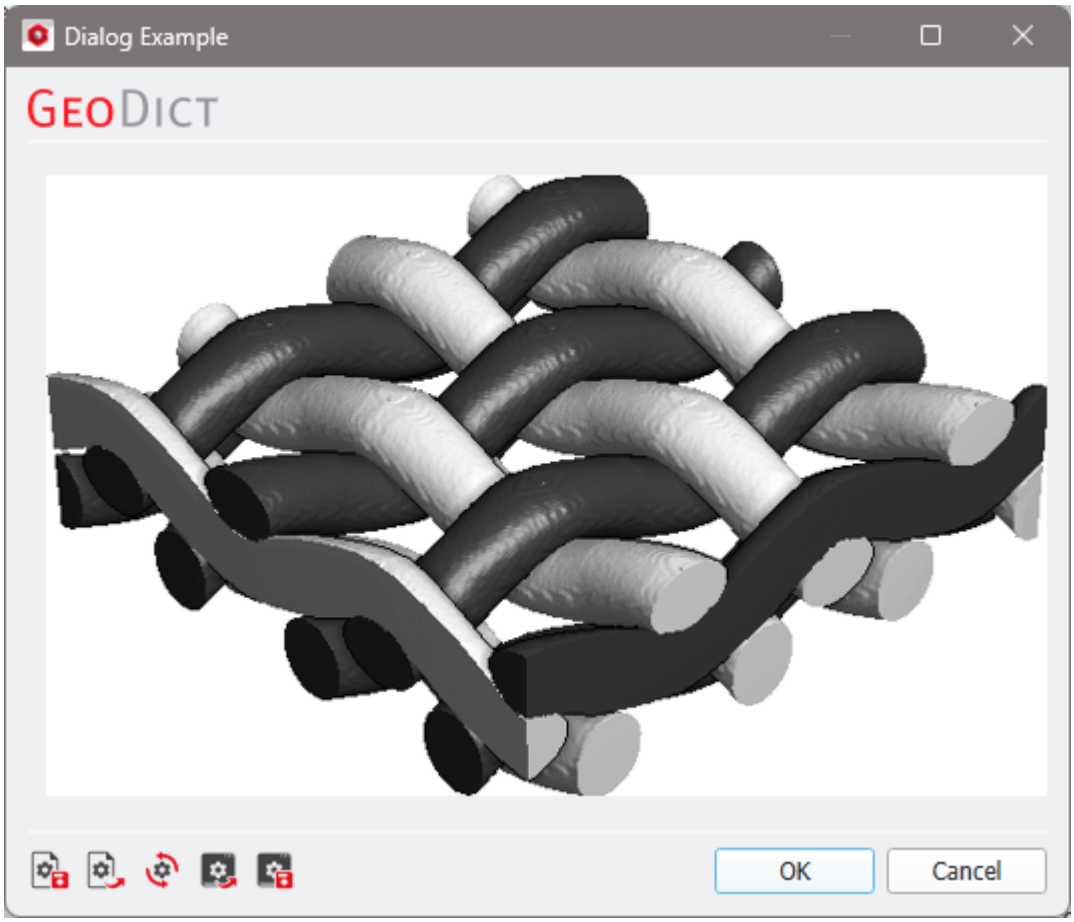
Add images to the dialog box. Before you can add them, they must be transformed to a 3D numpy array, which can for example be done using the Python packages PIL and numpy.

Input:

- image, 3D numpy array

Example:

```
dlg = gd.makeDialog("MyDialog",          # create a dialog object
"Dialog Example")                        # with a key and a title

from PIL import Image                    # import Python package
                                         # to edit images

import numpy as np                       # import Python package
                                         # to use arrays

image = Image.open("image.png")          # open desired image, if image
                                         # is not contained
                                         # in project folder,
                                         # complete file path
                                         # must be given

w,h = image.size                         # get size of image

image =                                  # resize image to fit in the
image.resize((500,round(500*h/w)))       dialog,
                                         # without changing aspect ratio

I = np.asarray(image)                    # transform image to a
                                         # 3D numpy array

dlg.addImage(I)                          # add image to the dialog

result = dlg.run()                       # open the dialog with all the
                                         # settings assigned to the dialog
                                         # variable, here dlg
```



.run

Execute the dialog.

Input: None

Example: see any other function above.

.saveSettings

Save the dialog settings as a *.gps file (GeoDict parameter settings) after calling run 110 .

Input:

- file path, string

Example: see below in the examples for interaction with dialog object 116

Input explanations for the sub functions:

- key

- Define a name for this argument to call the entered value within the python script

- description
  - this description is shown in the generated dialog box
- file type
  - define the file type for the input file. If set, only this file type can be found when browsing for the file.
- init
  - the init argument gives the initial value for the field (the built-in default)
- tooltip
  - the tooltip specifies a description string that is shown when the user hovers the mouse over the input field.
- min/max
  - min/max arguments restrict the range of input
- font size
  - set font size of text
- setBold
  - text is displayed in bold (True) or not (False). By default it is not bold.

All examples in one dialog

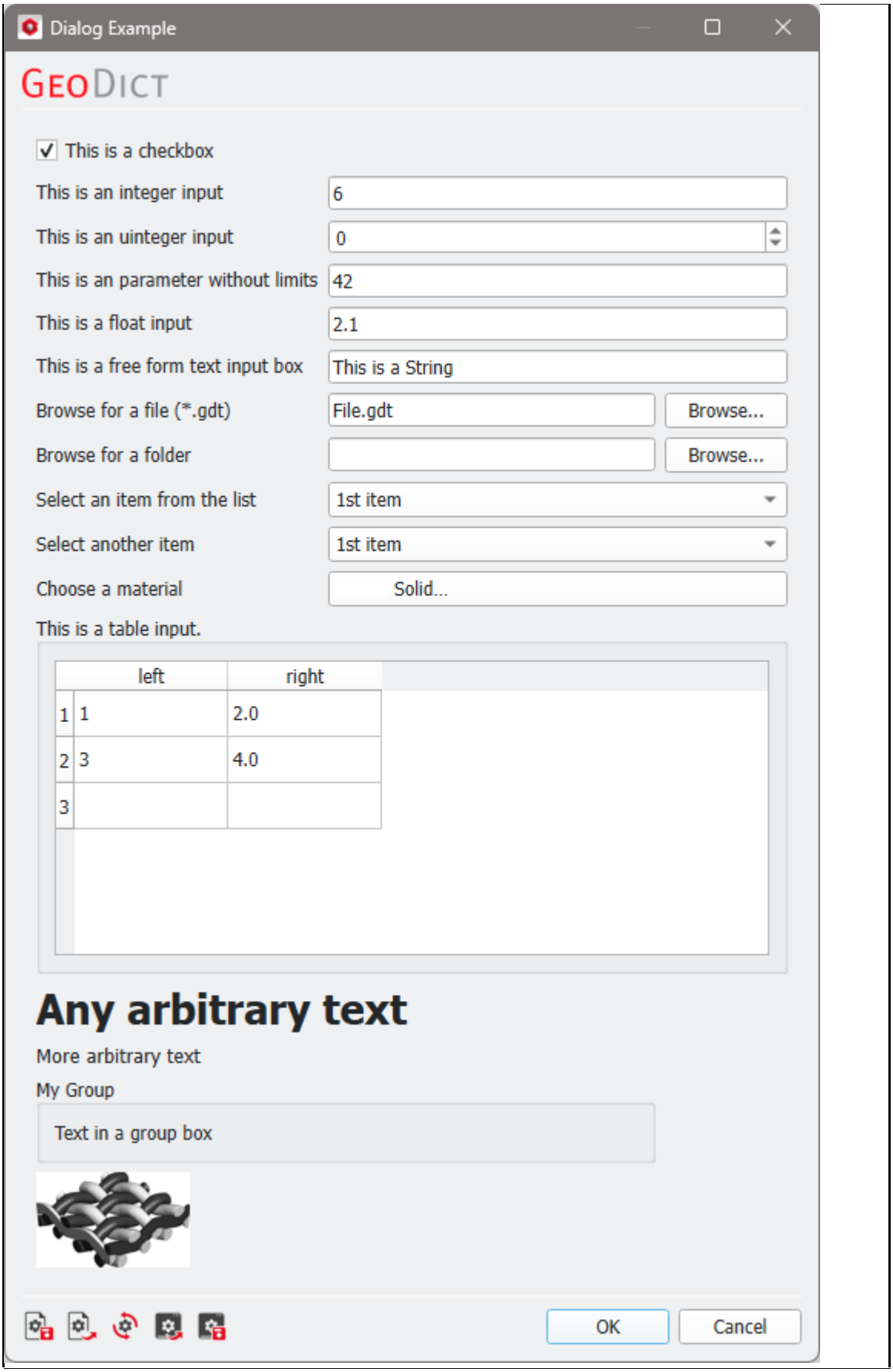
Example:

```
dlg = gd.makeDialog("MyDialog",      # create a dialog object
"Dialog Example")                    # with a key and a title



dlg.addBoolInput("myBooleanParamet   # create a check box,
er",                                 # with key, description,
  "This is a checkbox", init=True,   # default value and tooltip

  tooltip="Check this box or
not.")

dlg.addIntegerInput("myIntegerPara   # create an input field
meter",                              # for integer values with key,
  "This is an integer input",        # description, boundaries,
min=5, max=10,                       # default value and tooltip
  init=6, tooltip="Enter an
integer value.")

dlg.addUintegerInput("MyUintInput"   # create an input field
,                                    # for uinteger values with key,
```

```
  "This is an uinteger input",      # description, boundaries,
  min = -5, max = 5, init=0,        # default value and tooltip
  tooltip="Choose an integer
parameter")

dlg.addIntegerInput("myNewIntegerP  # create an input field
arameter",                          # for integer values with key,
  "This is an parameter without     # description, default value
limits",                            # and tooltip
  init=42, tooltip="Enter some
value here.")

dlg.addFloatInput("myFloatParamete  # create an input field
r",                                 # for float values with key,
  "This is a float input",          # description, boundaries,
  min = -3.5, max = 5.2, init=2.1,  # default value and tooltip

  tooltip="Enter a floating
value.")



dlg.addTextInput("myStringParamete  # create an input field
r",                                 # for text with key,
  "This is a free form text input   # description, default value
box",                               # and tooltip
  init="This is a String",
  tooltip="Enter text.")

dlg.addFileInput("myFileSelection"  # create an input field for
,                                   # gdt files with key,
  "Browse for a file", "gdt",       # description, default value
  init="File.gdt",                  # and tooltip
  tooltip="Browse for a gdt
file.")

dlg.addFolderInput("MyFolderInput"  # create an input field for
,                                   # folder paths with key,
  "Browse for a folder")            # and description

dlg.addComboInput("myComboBox",     # create a pull-down menu
  "Select an item from the list",   # with the given values for
  ["1st item", "2nd item", "3rd     # selection.
item"])                             # Key and description are given.
                                    # The returned value is the
                                    # index of the selected item,
                                    # e.g., 0 for the first item,
                                    # 1 for the second etc.

dlg.addComboInputString("ComboStri  # create a pull-down menu
ng",                                # with the given values for
  "Select another item",            # selection. Key and description
  ["1st item", "2nd item", "3rd     # are given.
item"])                             # The returned value is
                                    # the string of the
```

```
                                    # selected item

dlg.addMaterialInput("MyMaterialIn  # create an input button for
put",                               # browsing the material data
  "Choose a material")              # base. Key and description are
                                    given.

dlg.addTableInput("MyTableInput",   # create a table with two columns,
  "This is a table input.",         # one for integer input
  types = "int,float",              # and one for float.
  columnHeaders=["left","right"],   # Two default rows are given.,
  init=[[1,2.0],[3,4.0]])           # additionally to the mandatory
                                    # key, description, types and
                                    headers



dlg.addText("Any arbitrary text",   # add bold text to the
  20, True)                         # dialog with font size 20

dlg.addText("More arbitrary text",  # add text to the dialog
                                    # with font size 10
  10, False)                        # and not bold

dlg.beginGroup("My Group")          # start a group
                                    # to add a box

dlg.addText("Text in a group box")  # add any content inside
                                    # box, here we added
                                    # only some text

dlg.endGroup()                      # end the group



from PIL import Image               # import Python package
                                    # to edit images

import numpy as np                  # import Python package
                                    # to use arrays

image = Image.open("image.png")     # open desired image, if image
                                    # is not contained
                                    # in project folder,
                                    # complete file path
                                    # must be given

w,h = image.size                    # get size of image

image =                             # resize image to fit in the
image.resize((100,round(100*h/w)))  dialog,
                                    # without changing aspect ratio

I = np.asarray(image)               # transform image to a
                                    # 3D numpy array
```

```
dlg.addImage(I)                          # add image to the dialog


result = dlg.run()                       # open the dialog with all the
                                         # settings assigned to the dialog
                                         # variable, here dlg
```


Dialog Example
GeoDict
This is a checkbox
This is an integer input
6
This is an uinteger input
0
This is an parameter without limits
42
This is a float input
2.1
This is a free form text input box
This is a String
Browse for a file (*.gdt)
File.gdt
Browse...
Browse for a folder
Browse...
Select an item from the list
1st item
Select another item
1st item
Choose a material
Solid...
This is a table input.
left
right
1
1
2.0
2
3
4.0
3
Any arbitrary text
More arbitrary text
My Group
Text in a group box
OK
Cancel

Examples for interaction with dialog object

If the user clicks **Cancel**, result will be **None**.

Otherwise, result will be a dictionary containing the entered values, e.g.,

```
gd.msgBox("You have selected the file:"
  + result["myFileSelection"])                    # access a dialog value

dlg.saveSettings("MyFirstCustomDialogSettings.gps")  # save the entered settings
                                                     # as *.gps file
                                                     # (GeoDict parameter settings)
```

Graph Dialog Class

Create a GeoDict graph with the graph dialog class.

gd.makeGraphDialog

Graph dialogs allow displaying multiple graphs with multiple data sets per graph.

Input: None

Sub functions

.addGraph

Define the graph title and the axis legend labels.

Input:

- Graph title, string
- X-axis legend, string
- Y-axis legend, string

.addData

Add as many plots as desired by adding multiple **addData** lines.

Input:

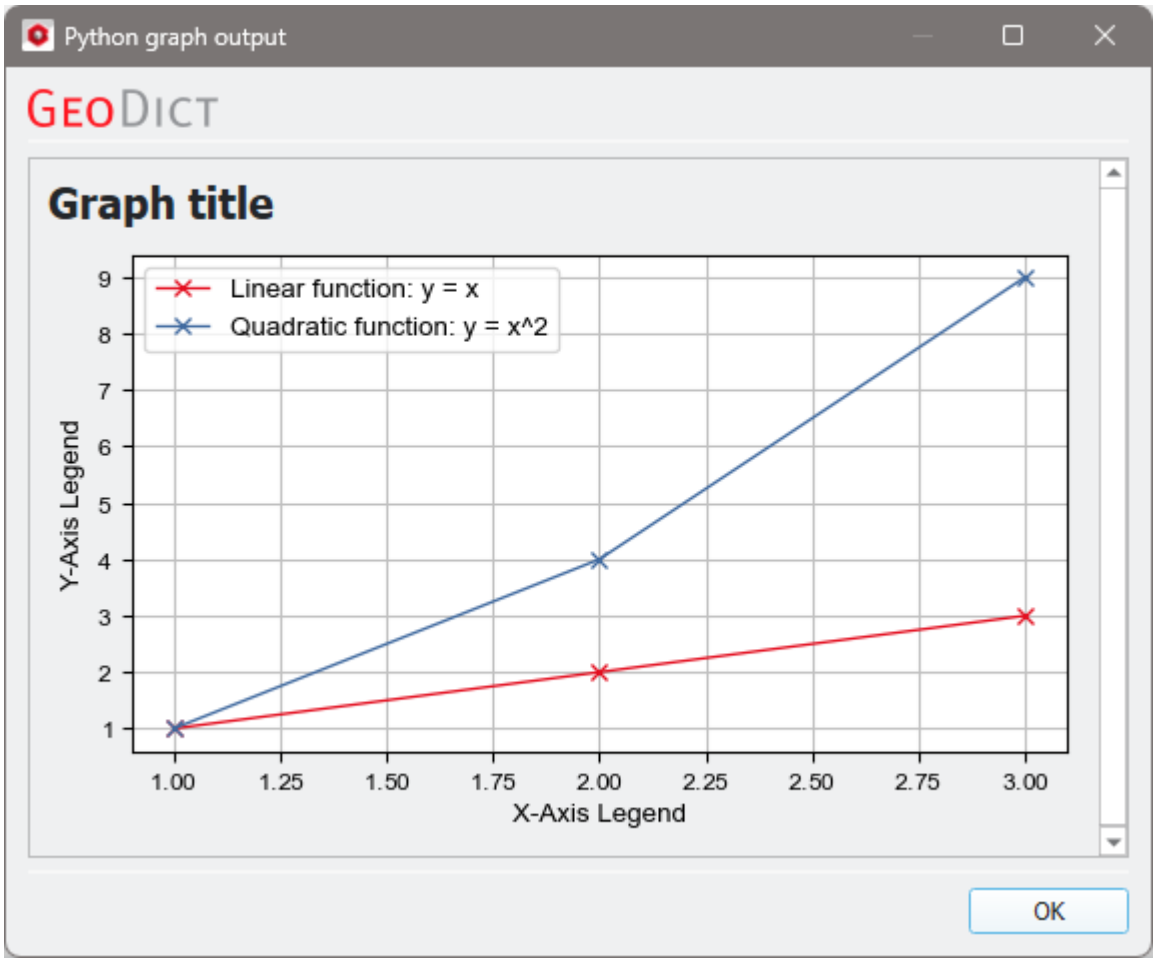
- X-values, list
- Y-values, list
- plot label, string

.run

When calling **run**, the graph dialog is displayed. By right-clicking in the plot the graphs offer the same features as the ones in the GDR visualization, e.g., the axes can be rescaled, the data can be exported as a CSV file using the context menu on each graph object, and the image can be saved as *.png.

Example

```
gDlg = gd.makeGraphDialog()                 # create a graph
                                            # dialog object

graph1 = gDlg.addGraph("Graph title",       # add a graph object
"X-Axis Legend", "Y-Axis Legend")           # with the given title,
                                            # x-axis legend
                                            # and y-axis legend

graph1.addData([1,2,3], [1,2,3],            # add a single dataset
"Linear function: y = x")                   # with the given x values,
                                            # y values and
                                            # legend to this graph

graph1.addData([1,2,3], [1,4,9],            # add another dataset
"Quadratic function: y = x^2")

gDlg.run()                                  # execute the
                                            # graph dialog
```



Visualization Functions

Get the current view status as dictionary or plot.

gd.getViewStatus

Returns the current view status (settings for rendering). It has the same format as the argument for the **GeoDict:SetViewStatus** command in Python files.

It is useful to change render settings based on the current settings.

Input:

- GeoDict version, string

Example:

For example, change the angle of the camera:

```
d = gd.getViewStatus("2026")          # get the current rendering
                                      settings

d["Camera"]["Camera3D"]               # change angle of camera
["Rotation"]=[38,22,-65]
```

```
gd.runCmd("GeoDict:SetViewStatus",  # update settings
d, "2026")
```

gd.get2DViewAsPlot

Returns the 2D view of the loaded structure as a Python dictionary. This dictionary can be used to plot the given slice in a custom GeoDict result file (*.gdr).

Input:

- view direction, integer
  - The view direction must be given as integer, where 0 = X, 1 = Y and 2 = Z.
- slice, integer
  - The slice in the 2D view of the visualization area in the given direction.
- orientation, bool
  - Determine, if the image orientation should be **Top to Bottom** (True) or **Bottom to Top** (False). These options are described with the other Camera 2D settings.

Example:

In the following example a result file is generated only containing a plot from the 50th slice of the loaded structure viewed in X-direction and bottom to top.

```
import gdr                              # import the module gdr to
                                        generate
                                        # custom result files

plotParameters =                        # get the current 2D view in X-
gd.get2DViewAsPlot(0,50,False)          direction
                                        # of slice 50 in bottom to top
                                        orientation

resultfile =                            # create custom result file
gdr.GDR("NewResultFile")                # NewResultFile.gdr

postParameters = {                      # define Python dictionary for
  'Plots' : {                           gdr
    'NumberOfPlots' : 1,
    'Plot1' : plotParameters}}

resultfile.postMap =                    # add post processing map to gdr
postParameters                          containing
                                        # the defined plot

resultfile.write()                      # write result file

gd.showGDR("NewResultFile.gdr")         # open result file in Result
                                        Viewer
```

ImportGeo-Vol Functions

There are several ImportGeo-Vol specific functions to create your own custom image processing filter. These functions do only work if a gray value image is loaded into ImportGeo-Vol. To load a gray value image, you need to run an **ImportGeo:GetGrayValueImage** command first.

Find examples on how to use these functions in the **ImportGeo** folder in the GeoDict installation directory. These functions are used to create your own custom image processing filter as described in ImportGeo-Vol.

gd.ImportGeoVol.getHistogram

Returns the histogram of the currently loaded image as a python list of tuples containing value and count each.

Input: None

Example:

In the following example the list is written into a *.csv file. If this file is opened with Excel, the gray values are to be found in the first column and the corresponding counts in the second column:

```
Histogram =                              # get list of tuples describing
gd.ImportGeoVol.getHistogram()           # the histogram and assign it
                                         # to the variable Histogram

file = open('Histogram.csv', 'w')        # open output file for writing (
                                         # create new file with the given
                                         name,
                                         # if file does not exist)

file.write('Value,Count\n')               # write titles for columns in csv
                                         file

for i in Histogram:                      # loop over all tuples i of
                                         Histogram

  file.write(f'{i[0]},{i[1]}\n')         # write values and counts
                                         # into the csv file

file.close()                             # close the csv file
```

gd.ImportGeoVol.getNewImage

Creates a new 3D gray value image matching the size and bit depth as the original image and returns it as a numpy array. Only used in ImportGeo custom python image filters, not in regular macros.

Input: None

gd.ImportGeoVol.getNewImageDimensions

Returns the current gray value image size in voxels in the desired direction.

Input:

- direction, integer
  - 0 for X-direction, 1 for Y-direction, 2 for Z-direction

Example:

```
nx =                                    # get number of voxels in
gd.ImportGeoVol.getNewImageDimensi      # X-direction (direction 0)
ons(0)

gd.msgBox(f"In X-direction there        # show message box
are {nx} voxels.")
```

gd.ImportGeoVol.getNewImageResized

Creates a new 3D gray value image with the entered dimensions. Only used in ImportGeo custom python image filters, not in regular macros.

Input:

- nx, integer
  - number of voxels in x-direction
- ny, integer
  - number of voxels in y-direction
- nz, integer
  - number of voxels in z-direction
- is 16 bit, bool
  - the image can have 16 bit (True) or 8 bits (False).

gd.getOriginalImage

Returns the currently loaded gray value image as a 3D 8-bit or 16-bit numpy array. Only used in ImportGeo custom python image filters, not in regular macros.

Input: None

gd.ImportGeoVol.getOtsuThreshold

Returns the threshold based on OTSU's method of the currently loaded image as an integer.

Input: None

Example:

```
OTSU =                                  # get threshold and assign
gd.ImportGeoVol.getOtsuThreshold()      # it to variable OTSU

gd.msgBox(f"OTSU threshold is           # show message box
{OTSU}")
```

gd.ImportGeoVol.getMultiOtsuThreshold

Returns the thresholds based on OTSU's method of the currently loaded image as list.

Input: None

Example:

```
OTSU =                                  # get threshold list and
gd.ImportGeoVol.getMultiOtsuThresh      # assign it to variable OTSU
old()

gd.msgBox(f"The OTSU thresholds         # show message box
are {OTSU}")
```

gd.ImportGeoVol.getVoxelLength

Returns the currently in ImportGeo-Vol set voxel length.

Input: None

Example: For an example, see below under the **setVoxelLength** 122 command.

gd.ImportGeoVol.setVoxelLength

Changes the currently in ImportGeo-Vol set voxel length to the specified value. This command has no return value.

Input:

- Voxel length, float

Example:

```
vl_old =                                # get voxel length of
gd.ImportGeoVol.getVoxelLength()        # currently loaded gray value
                                        # image and assign it to
                                        # variable vl_old

gd.ImportGeoVol.setVoxelLength(1e-      # set voxel length of
6)                                      # currently loaded gray
                                        # value image to 1µm
```

```
vl_new =                                # assign new voxel length
gd.ImportGeoVol.getVoxelLength()        # to variable vl_new

gd.msgBox(f"The voxel length was        # show message box
changed from {vl_old} to
{vl_new}")
```

gd.ImportGeoVol.getRotationSuggestion

The command returns the rotation suggested for the loaded gray value image.

Input:

- full image, bool
  - Suggest full image (True) or plane (False)
- threshold, integer (optional)

Example:

```
Rot =                                   # get rotation suggestion
gd.ImportGeoVol.getRotationSuggest      # for suggest plane and
ion(False)                              # assign it to variable
                                        # rotation.
                                        # Threshold is omitted -
                                        # default to automatic
                                        # threshold

Rotation_args =                         # get Built-in Defaults
gd.getBuiltinDefaults("ImageProces      # for the Python dictionary
sing:Rotation")                         # of the GeoDict command
                                        # Rotation and assign the
                                        # dictionary to variable
                                        # Rotation_args

Rotation_args['Phi']   = Rot[0]         # assign the rotation values
                                        # to the corresponding keys
                                        # in the rotation dictionary

Rotation_args['Theta'] = Rot[1]

Rotation_args['Psi']   = Rot[2]

gd.runCmd("ImageProcessing:Rotatio      # rotate the gray value image,
n", Rotation_args, '2026')              # version is omitted -
                                        # default to latest
```

Particle Functions

For the following particle specific functions a visualization of particles (from FilterDict or AddiDict) must be loaded.

gd.getParticlesInfo

Get the current view status as dictionary or plot.

Returns a Python dictionary containing the number of batches and the maximal and minimal batch animation times.

Input: None

Example:

```
Info = gd.getParticlesInfo()              # assign the Particles Info
                                          # dictionary to the variable Info

Num  = Info['NumberOfBatches']            # assign the number of batches
                                          # to the variable Number

gd.msgBox (f"The number of batches is     # show message dialog
{Num}.")
```

gd.getParticles

Returns the Particles object which gives access to currently loaded particle data. To obtain the data the **GeoParticles** class is used. It works only in combination with this command.

Input:

- GeoDict version, string

Example: For an example, see below in the subfunctions of the getParticles class.

Subfunctions

.getBatchInfo

Returns information about a batch of particles as a Python dictionary. The resulting dictionary contains:

- "minTime": start time of batch
- "maxTime": end time of batch
- "minRadius": minimal particle radius in batch
- "maxRadius": maximum particle radius in batch
- "particleIds": list of particle IDs present in this batch

This command only works in combination with the **gd.getParticles** 124 command.

Input:

- batch index, integer

Example:

```
particles =                          # assigns the particles object
gd.getParticles("2026")              # to the variable particles

BatchInfo =                          # dictionary containing batch
particles.getBatchInfo(1)            # info is assigned to the
                                     # variable BatchInfo

print(BatchInfo["minTime"])          # prints the start time
                                     # of batch 1 to the console
```

.getDiameter

Returns the (interpolated) particle diameter at a given time. This command only works in combination with the gd.getParticles command.

Input:

- batch index, integer
- particle index, integer
- time, float

Example:

```
particles =                          # assigns the particles object
gd.getParticles("2026")              # to the variable particles

Diameter =                           # the diameter of the particle
particles.getDiameter(1, 2000, 1)    # in batch 1 with particle ID
                                     2000
                                     # at time 1 is assigned to
                                     # the variable Diameter


print(f"The particle diameter is     # prints the diameter to the
{Diameter}m")                        console
```

.getDiameters

Computes the (interpolated) particle diameters with a given step size. Sampling starts at “minTime” and increments by step size up to “maxTime”. Returns a list of tuples (time, radius). This command only works in combination with the gd.getParticles command. The command makes sense, only when the particle with the given particle index changes its diameter over time. Otherwise, an empty list is returned.

Input:

- batch index, integer

- particle index, integer
- time step, float

.getLoadedBatchIndices

Returns a list of valid particle batches that are currently loaded in memory. This command only works in combination with the gd.getParticles command.

Input: None

Example:

```
particles =                          # assigns the particles object
gd.getParticles("2026")              # to the variable particles

Batches =                            # assign the list of the batch
particles.getLoadedBatchIndices()    # indices to the variable Batches

print(Batches)                       # prints the list to the console
```

.getMultiplicities

Computes the (interpolated) particle multiplicity with a given step size. Sampling starts at "minTime" and increments by step size up to "maxTime". Returns a list of tuples (time, multiplicity). This command only works in combination with the gd.getParticles command.

Input:

- batch index, integer
- particle index, integer
- time step, float

Example:

```
particles =                          # assigns the particles
gd.getParticles("2026")              # object to the variable
                                     # particles

M = particles.getMultiplicities(1,   # assign the list of
2000, 0.0001)                        # tuples to the
                                     # variable M

print(f"In batch 1 the particle      # prints the values
2000 has multiplicity {M[1][1]} at   # of the second tuple
time {M[1][0]}.")                    # to the console
```

.getMultiplicity

Computes the (interpolated) particle multiplicity at a given time. This command only works in combination with the gd.getParticles command.

Input:

- batch index, integer
- particle index, integer
- time, float

Example:

```
particles =                          # assigns the particles
gd.getParticles("2026")              # object to the variable
                                     # particles

M = particles.getMultiplicity(1,     # assign the multiplicity
2000, 0.0001)                        # to the variable M

print(f"In batch 1 the particle      # prints the multiplicity
2000 has multiplicity {M} at time    # to the console
0.0001.")
```

.getParticleInfo

Returns information about a particle inside a batch as a Python dictionary. This command only works in combination with the gd.getParticles command.

The resulting dictionary contains:

- "minTime", "maxTime": start/end time of particle trajectory
- "material_id": the material ID of the particle
- "type": type index of the particle
- "status_code": numerical status of the particle
- "status": human-readable interpretation of particle status (e.g., "EXIT_OUTFLOW", "TRAPPED_SIEVING")
- "end_material_id": if status is "HIT_END_MATERIAL", this contains the material id which the particle hit
- "is_ghost": True if ghost particle
- "times": time values for individual sample points along the trajectory
- "positions": particle position for each time
- "radii": particle radius for each time or single value if not time-dependent
- "multiplicities": particle multiplicity for each time

Input:

- batch index, integer
- particle index, integer

Example:

```
particles =                              # assigns the particles
gd.getParticles("2026")                  # object to the variable
                                         # particles

Info =                                   # assign the material ID
particles.getParticleInfo(1, 3500)       # to the variable M

M_ID = Info["material_id"]

print(f"The material ID of the           # prints the material ID
particle is {M_ID}.")                    # to the console
```

.getPosition

Returns the (interpolated) particle position at a given time. This command only works in combination with the gd.getParticles command.

Input:

- batch index, integer
- particle index, integer
- time, float

Example:

```
particles =                              # assigns the particles object
gd.getParticles("2026")                  # to the variable particles

Pos = particles.getPosition(1,           # assign the position of a
3500, 0.0001)                            # particle to the variable Pos

print(f"The position of the              # prints the position
particle is {Pos}.")                     # to the console
```

.getPositions

Computes the (interpolated) particle positions with a given step size. Sampling starts at "minTime" and increments by step size up to "maxTime". Returns a list of tuples (time, position). This command only works in combination with the gd.getParticles command.

Input:

- batch index, integer
- particle index, integer

Example:

```
particles =                              # assigns the particles object
gd.getParticles("2026")                  # to the variable particles

Pos = particles.getPositions(1,          # assign the list of tuples
3500, 0.0001)                            # (time, position) of a particle
                                         # to the variable Pos

print(f"The position of the              # prints the second
particle is {Pos[0][1][1]} at time       # (time, position) tuple
{Pos[0][1][0]}.")                        # to the console
```

.getPositionsAtTime

Computes all (interpolated) particle positions at a given time. This command only works in combination with the gd.getParticles 124 command.

Input:

- batch index, integer
- time, float

Example:

```
particles =                              # assigns the particles object to
gd.getParticles("2026")                  # the variable particles

Pos =                                    # assign the list of positions of
particles.getPositionsAtTime(1,          # all particle at the given time
0.0001)                                  # to the variable Pos

print(f"The position of the              # prints positions to the console
particles are {Pos}.")
```

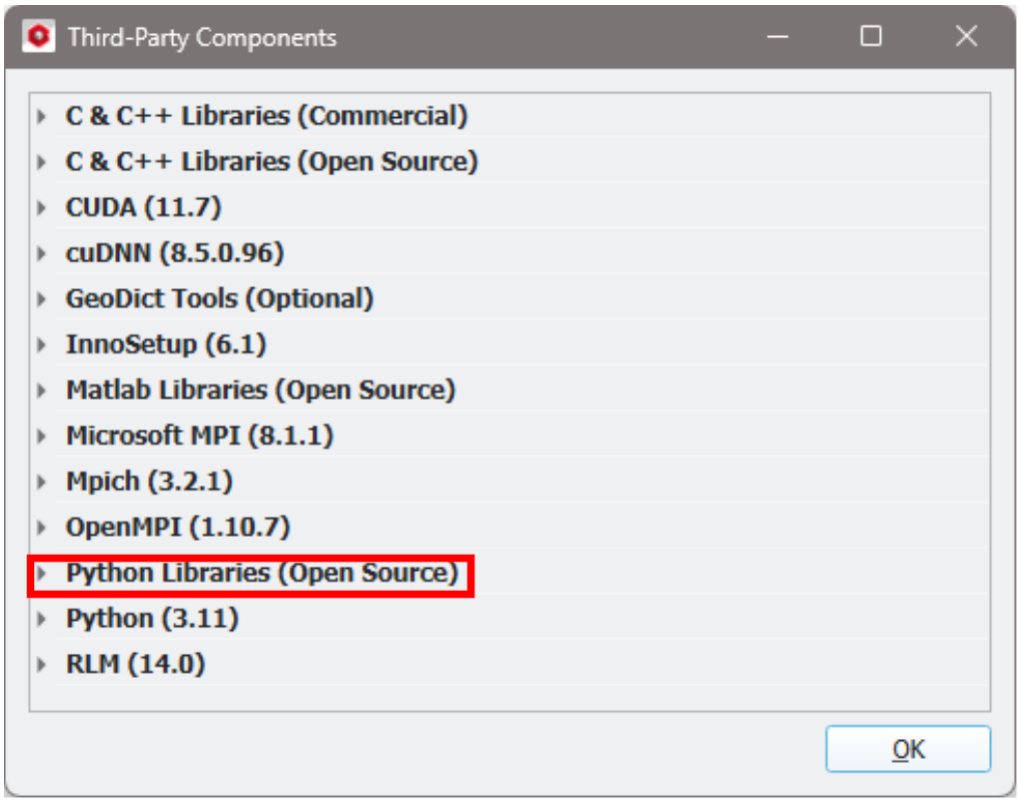
1.7.2 Shipped Python Modules

In addition to the API provided by the gd object, GeoDict also includes some Python packages (inside the gd folder), which are useful for reading/writing GeoDict file format and some other APIs, as for example the **stringmap** module (stringmap.py) which can be used to parse GeoDict key/value text file formats such as GDR files and the gdr module to create custom GeoDict result files 142.

The following table shows the most important Python libraries, that are shipped with GeoDict. To use them in a macro, import the respective module in the first lines of the macro, as shown above with the module **stringmap** 137.

Library	Description
Matplotlib	Graph plotting and data visualization library.
NumPy	Fast numerical calculations. The GeoPy API uses NumPy data types for accessing structures and volume fields.
Pillow	Library to read, write, and manipulate images.
XlsxWriter	Create Excel files from GeoPy.
python-pptx	Library to create PowerPoint slides. Note: GeoDict provides a simplified wrapper API in the gd_ppt namespace.
SciPy	Library for scientific & numerical computation (integration, interpolation, optimization, linear algebra, statistics).
lxml	XML & HTML processing library.
psutil	Library for accessing information about the operating system and currently running processes.

The complete list of all Python packages shipped with GeoDict can be found by selecting **Help → Acknowledgments** from the menu bar and unfolding **Python Libraries (Open Source)** in the opening dialog.



If a module is needed which is not shipped with GeoDict, it can be installed[32].

1.7.3 PowerPoint Report Generation

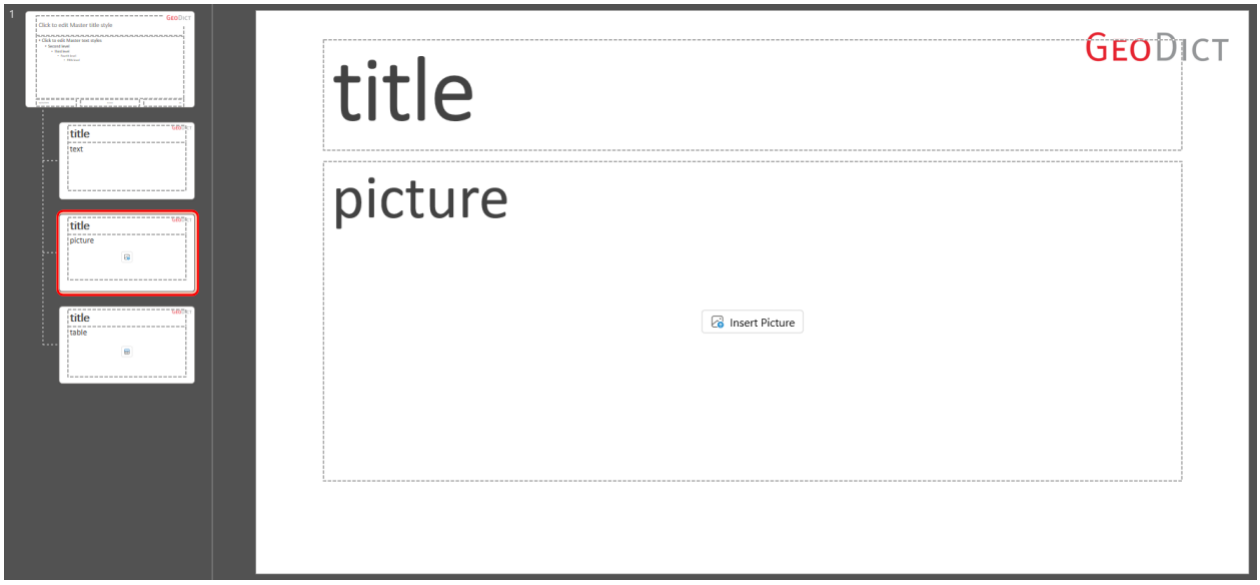
GeoDict includes a simplified wrapper API to create PowerPoint files. This is particularly useful, if the same workflow is repeated often with different parameters in an automatic parameter study and the results should be presented in a PowerPoint report. In this way, **gd_ppt** provides a simple possibility to compare the results as desired.

The general idea is to prepare an empty PowerPoint file, containing only slide masters, which is loaded with the **gd_ppt** library from a Python file. For each slide to be generated, an empty layout master slide is selected and added to the presentation. Then, the placeholders are replaced by actual content. The supported content types are **text**, **pictures**, **movies**, and **tables**. The placeholders are identified by the text inside the placeholder.

To prepare your own template, save a copy of your own corporate design PowerPoint template, containing only master slides. In PowerPoint and then change to the master view by selecting **View** → **Slide Master** from the toolbar.

The layout master slides are organized under an overall **Theme Master Slide**. Change only the needed **Layout Masters** by replacing the text in the needed placeholder by a single, rememberable name, e.g., title or picture.

The following screenshot shows layout masters with placeholders. The ***slide indices*** are shown here with red numbers. Observe that the slide counting starts with zero.



In the figure above, the selected example layout master with index 1 has two placeholders called **title** and **picture**.

In the examples given for the sub functions, only one slide was added for each PowerPoint report. Of course, the number of slides added to a report is not limited. Add as many slides as desired between the lines `rep = gd_ppt.ReportGenerator` and `rep.save`.

You can download the PowerPoint template and the GeoPy script to recreate the shown examples.

Download PowerPointReportExample.zip

The `gd_ppt` library is loaded at the beginning of a Python file with the command `import gd_ppt` and contains the following commands and sub-functions:

gd_ppt.reportgenerator

Opens the template PowerPoint file.

Input:

- template file, string

Example: see any of the sub functions

Sub functions

.add_slide

Adds a slide with the style defined by the Layout Master with the given index.

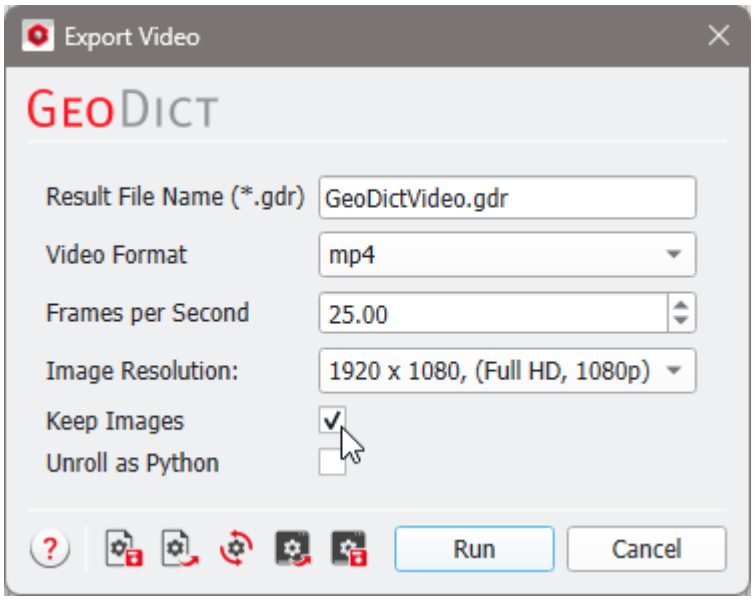
Input:

- slice index, integer

Example: see any of the other sub functions

.add_movie

Replaces the given picture placeholder by a movie. For the movie, a thumbnail is needed, that is shown before the movie is played back. Therefore, a folder with the same name as the movie (e.g., example, if the movie is named "example.mp4") must be located in the same folder as the movie and should contain the folder **images** with at least one picture.



This folder is automatically generated if a video is generated with GeoDict and **Keep Images** is checked. For the add_movie command, a ***picture placeholder*** is needed.

Input:

- picture placeholder, string
- movie file path, string

Example:

```
import gd_ppt                          # import PowerPoint API
                                       # wrapper library

rep =                                  # create a report generator
gd_ppt.ReportGenerator("example_te     # based on master slides in
mplate.pptx")                          # "example_template.pptx"

sl = rep.add_slide(1)                  # create a slide based on
                                       # the second layout master
                                       # (index number 1)
```

```
sl.add_text("title", "Movie Example")          # fill the placeholder title
                                               # with the text Movie Example

sl.add_movie("picture", "example_movie.mp4")   # fill the placeholder picture
                                               # with the movie example from
                                               # the project folder

rep.save("report_example.pptx")                # save the PowerPoint
                                               # presentation with the name
                                               # report_example.pptx
```

.add_picture

Fills a picture in the given picture placeholder. For this command, a ***picture placeholder*** is needed.

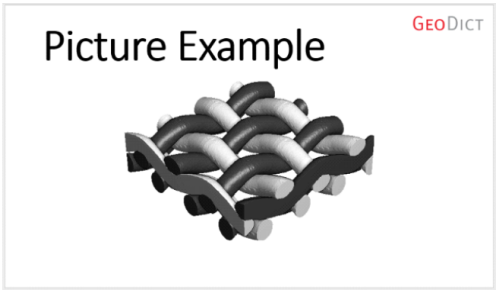
Input:

- picture placeholder, string
- picture file path, string

Example:

```
import gd_ppt                                  # import PowerPoint API
                                               # wrapper library

rep = gd_ppt.ReportGenerator("example_template.pptx")  # create a report generator
                                               # based on master slides in
                                               # "example_template.pptx"

sl = rep.add_slide(1)                          # create a slide based on
                                               # the second layout master
                                               # (index number 1)

sl.add_text("title", "Picture Example")        # fill the placeholder
                                               # title with the text
                                               # Picture Example

sl.add_picture("picture", "example_picture.png")  # fill the placeholder
                                               # picture with the
                                               # picture example from
                                               # the project folder

rep.save("report_example.pptx")                # save the PowerPoint
```

```
                                        # presentation with
                                        # the name
                                        # report_example.pptx
```



.add_table

Transforms a list into a table and adds it to the given placeholder. The headers and the font size are optional. If both headers are given, the vertical header has to contain one additional entry for the horizontal header line. For the add_table command, a ***table placeholder*** is needed.

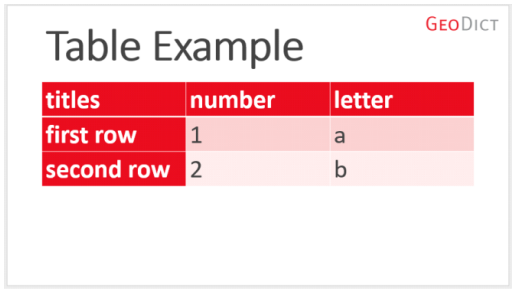
Input:

- table placeholder, string
- horizontal_header, list
- vertical_header, list
- font_size, float

Example:

```
import gd_ppt                           # import PowerPoint API
                                        # wrapper library

rep =                                   # create a report generator
gd_ppt.ReportGenerator("example_te      # based on master slides in
mplate.pptx")                           # "example_template.pptx"

sl = rep.add_slide(2)                   # create a slide based on
                                        # the third layout master
                                        # (index number 2)

sl.add_text("title", "Table             # fill the placeholder title
Example")                               # with the text Table Example

h_h = ["number", "letter"]              # assign a list for the
                                        # horizontal header to the
                                        # variable h_h

v_h = ["titles", "first row",           # assign a list for the
"second row"]                           # vertical header to the
                                        # variable v_h
```

```
table = [[1, "a"], [2, "b"]]          # assign a list for a 2x2
                                      # table to the variable
                                      # table with the entries 1
                                      # and a in the first row
                                      # and 2 and b in
                                      # the second row

sl.add_table("table", table,          # fill the placeholder table with
horizontal_header = h_h,              # the defined table and the
  vertical_header = v_h, font_size    # headers h_h and v_h,
= 50)                                 # and font size 50

rep.save("report_example.pptx")       # save the PowerPoint
                                      # presentation with the name
                                      # report_example.pptx
```



.add_text

Fills a text placeholder with text in the given font size. The font size is optional. If omitted, the resulting font size will be the same as used in the placeholder. For this command a ***text placeholder*** is needed.

Input:

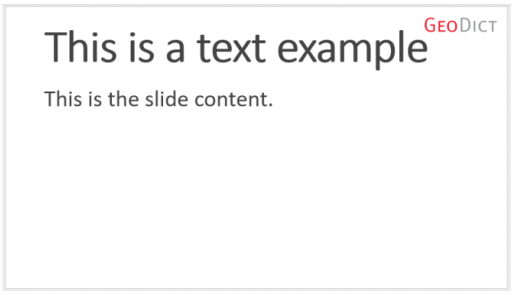
- text placeholder, string
- text, string
- font_size, float

Example:

```
import gd_ppt                         # import PowerPoint API
                                      # wrapper library

rep =                                 # create a report generator
gd_ppt.ReportGenerator("exampl        # based on master slides in
e_template.pptx")                     # "example_template.pptx"

sl = rep.add_slide(0)                 # create a slide based on
                                      # the first layout master
                                      # (index number 0)
```

```
sl.add_text("title", "This is a text example")    # fill the placeholder title
                                                  # with the text
                                                  # This is a text example.
                                                  # Font size is omitted

sl.add_text("text", "This is the slide content.", font_size = 45)    # fill the placeholder text
                                                                     # with the text
                                                                     # This is the slide content
                                                                     # in font size 45

rep.save("report_example.pptx")                   # save the PowerPoint
                                                  # presentation with the
                                                  # name report_example.pptx
```

The result of this example is a PowerPoint presentation containing the single slide shown on the right. The first picture shows the corresponding layout master from the template file with index 0. All placeholders have been replaced by actual content, e.g., **title** was replaced by **This is a text example**.



.save

Saves the PowerPoint presentation under the given name.

Input:

- file name, string

Example: see any of the other sub functions

1.7.4 Access to Result Files

Accessing GeoDict result files (*.gdr) from Python macros is possible with the stringmap module. Use it to parse GeoDict key/value text file formats such as GDR files. Stringmaps represent a hierarchical key/value data structure, like a nested dictionary.

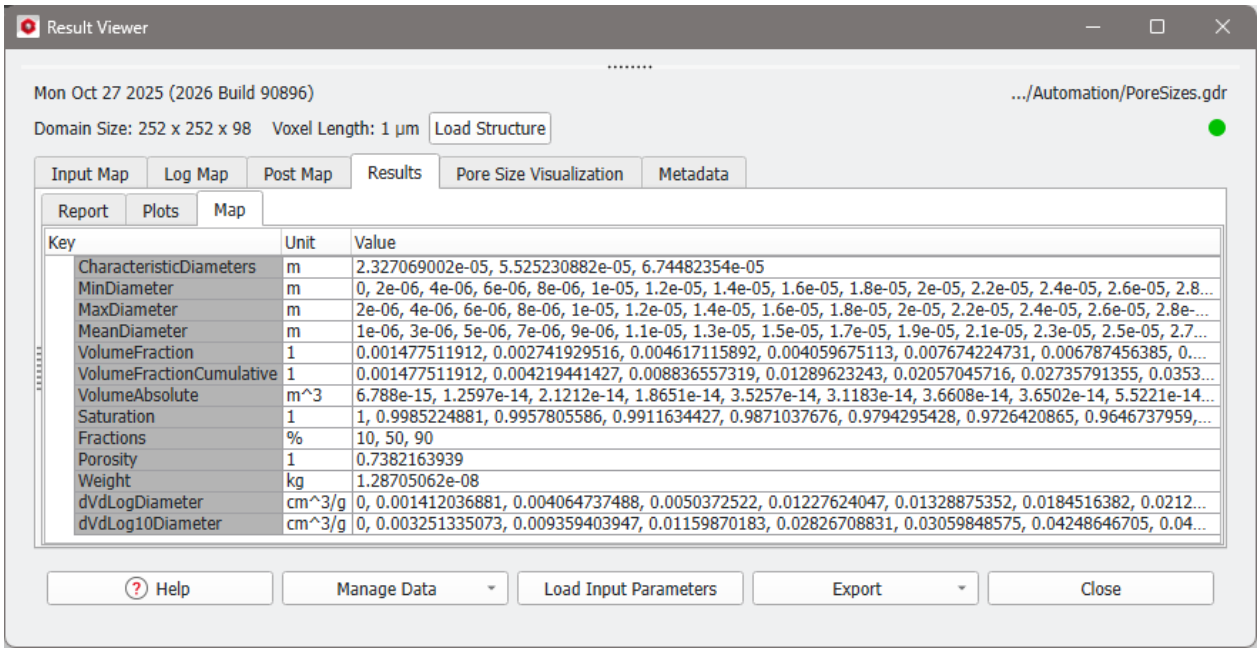
General Example

An example of usage, assuming a geometric pore size distribution (Granulometry) was run with PoroDict and the result file was saved as PoreSizes.gdr:

```
import stringmap                                  # The module stringmap is
```

```
                                        # loaded in the beginning.

gdrPath = "PoreSizes.gdr"               # assign GeoDict result file
                                        # PoreSizes to variable gdrPath

gdr = stringmap.parseGDR(gdrPath)       # read and parse the GDR
                                        # file into a string map
                                        # object and assign it to
                                        # variable gdr

gdr.push("ResultMap")                   # move into the result map
                                        # Alternatively, you can omit
                                        # this push and write
                                        # "ResultMap:MaxDiameter"
                                        # in the following line

maxDiameters       =                    # get the list values called
gdr.getList("MaxDiameter")              # "MaxDiameter" and
volFracsCumulative =                    # "VolumeFractionCumulative"
gdr.getList("VolumeFractionCumulative   # from the result map in the GDR
")                                      # To get other types of values
                                        # use one of the following methods:
                                        # gdr.getBool(key), gdr.getInt(key),
                                        # gdr.getDouble(key)
                                        # getList always returns a list
                                        # of strings, however

maxDiameters = [float(x) for x in       # do the following to convert
maxDiameters]                           # the string lists to numerical
                                        values

volFracsCumulative = [float(x) for x    # convert each list entry from a
in volFracsCumulative]                  # string to a floating point value

gdr.pop()                               # go back to the root of the tree
                                        # only needed when push was used
```

To find the right keys open a result file in the **GeoDict Result Viewer** by selecting **File** → **Open Results (*.gdr)** from the menu bar and move to the desired map tab, here the **Results – Map** subtab. The **Input Map** ("InputMap"), the **Log Map** ("LogMap"), the **Post Map** ("PostProcessingMap") and the **Parameter Map** under the **Metadata** tab ("ParameterMap") can be accessed in the same way.



.parseGDR

Parses the given GeoDict result file (*.gdr) and returns it as a map. The values are accessible by entering the right key in square brackets. For the right key, look in the result file as described above. The result file map has a tree like structure. The different levels of subtrees are separated by colon ":".

Input:

- gdr file path, string

Example:

```
import stringmap                          # import the stringmap module

gdrPath =                                 # define the file path of the
"C:/Automation/PoreSizes.gdr"             # gdr file to consider"

gdr = stringmap.parseGDR(gdrPath)         # parse the gdr file

weight = gdr["ResultMap:Weight"]          # get the value for weight

print(weight)                             # print the resulting values
```

.push

All operations following this command only consider the given subtree.

Input:

- key, string

Example:

For an example see the first example above[137].

.pop

After push was used, this command leads back to the root of the tree.

Input: None

Example:

For an example see the first example above[137].

.getUnit

Returns the unit for the given key as a string.

Input:

- key, string

Example:

```
import stringmap                          # import the stringmap module

gdrPath =                                 # define the file path of the
"C:/Automation/PoreSizes.gdr"             # gdr file to consider

gdr = stringmap.parseGDR(gdrPath)         # parse the gdr file

weight_unit = gdr.getUnit("ResultMa       # get the unit for weight

weight = gdr.getDouble("ResultMap:W       # get the value for weight
                                          # as number

print(weight, weight_unit)                # print the resulting values
```

.getDouble

Returns the value for the given key as floating point number.

Input:

- key, string

Example:

For an example, see above for getUnit[140].

.getInt

Returns the value for the given key as integer number.

Input:

- key, string

.getBool

Returns the value for the given key as bool (True or False).

Input:

- key, string

.getList

Returns the value for the given key as list.

Input:

- key, string

.hasKey

Checks if the given key is contained in the considered tree and returns True or False accordingly.

Input:

- key, string

Example:

For an example see below for getFullKey 141.

.getFullKey

Returns the full key path for the given key, e.g., for Porosity in the result file above it would return ResultMap:Porosity.

Input:

- key, string

Example:

```
import stringmap                        # import the stringmap module

gdrPath =                               # define the file path of the
"C:/Automation/PoreSizes.gdr"           # gdr file to consider"

gdr = stringmap.parseGDR(gdrPath)       # parse the gdr file

gdr.push("ResultMap")

if gdr.hasKey("Weight"):                # if the key "Weight" is
```

```
                                    # contained in the gdr the
                                    # following indented section
                                    # is executed

  key_path =                        # get the full key
gdr.getFullKey("Weight")            # path for "Weight"

  print(key_path)                   # print the full key
                                    # if the key is not in the gdr,
                                    # execute the following
                                    # indented section

else:                               # print message in
  print ("The key is invalid.")     # the GeoDict console
```

.toFile

Saves the stringmap to the given file. This offers the possibility to change result files.

Input:

- new result file path, string

Example:

In the following example, the value for "Weight" is changed and the new map is saved to a new result file.

```
import stringmap                    # import the stringmap module

gdrPath =                           # define the file path of the
"C:/Automation/PoreSizes.gdr"       # gdr file to consider"

gdr = stringmap.parseGDR(gdrPath)   # parse the gdr file

weight =                            # get the value for weight
gdr.getDouble("ResultMap:Weight")   # as number

weight_in_mg = weight*1000000       # transform weight from kg to mg

gdr["ResultMap:Weight"] =           # set the weight to 0.0128 mg
[weight_in_mg, "mg"]                # instead of 1.28e-10 kg.

gdr.toFile("NewResultFile.gdr")     # save the new map
                                    # under a new name"
```

1.7.5 Create Custom Result Files

GeoDict includes an API to create custom result files (*.gdr). This is particularly useful, if the same workflow is repeated often with different parameters in an automatic parameter study and the results should be presented in the GeoDict **Result Viewer**. In this way, the library **gdr** provides a simple possibility to compare the results as desired. For more details about result files refer to the Result Viewer User Guide.

The gdr library is loaded at the beginning of a Python file with the command **`from gdr import GDR`** and contains the following commands:

Basic functions

GDR.createEmptyResults

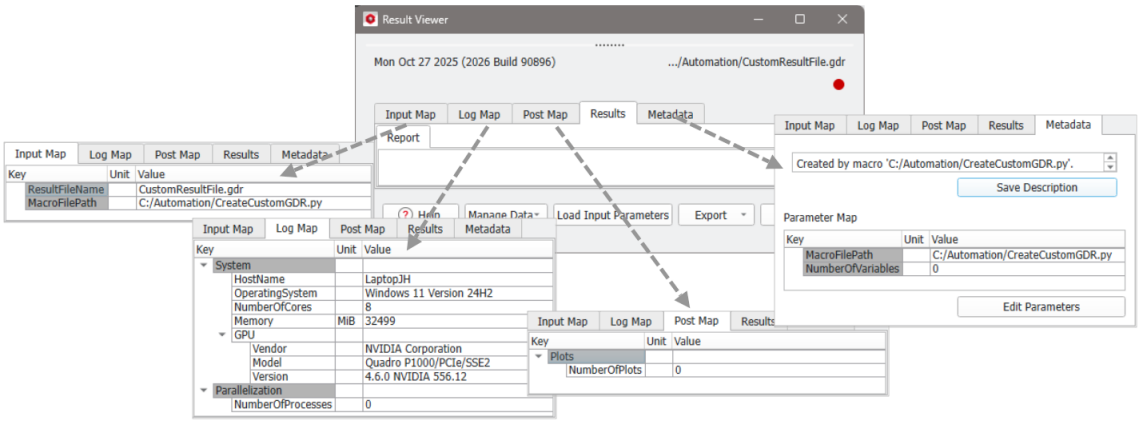
Creates an empty *.gdr file and a result folder with the given name. Start with this command to create a custom GeoDict result file already containing input map, log map, post map and results tabs. Additionally, the GeoDict project folder is changed to the result folder automatically.

Input:

- gdr file name, string
- GeoDict release, string

Example:

```
from gdr import GDR                          # import gdr library

gdrf =                                       # open a result file with the
GDR.createEmptyResults("CustomResu           # name CustomResultFile.gdr
ltFile","2026")                              # and assign it to the
                                             # variable gdrf
                                             # (GeoDict result file)
```



.saveResults

Saves the results to the result file. Define a result map and the report. This command saves the result file with all settings defined by the commands shown in this chapter used in the lines between createEmptyResults 143 and **saveResults**.

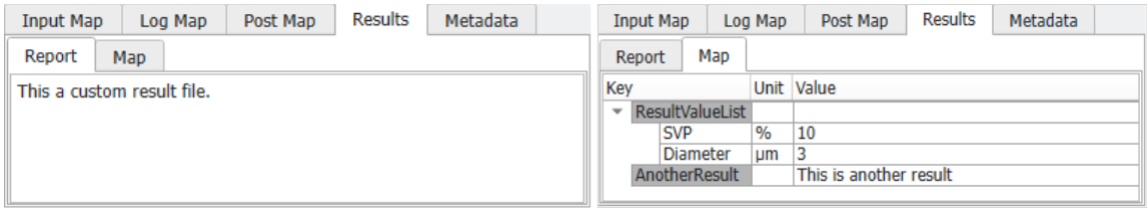
Input:

- result map, Python dictionary

- report, string
- GeoDict release, string

Example:

```
from gdr import GDR                          # import gdr library

gdrf =                                       # open a result file with
GDR.createEmptyResults("CustomResu           # name CustomResultFile.gdr
ltFile","2026")                              # and assign it to the
                                             # variable gdrf

report = "This a custom result               # define text for the
file."                                       # Report tab and assign it
                                             # to variable report

resultMap = {                                # define content for the
  'ResultValueList' : {                      # map subtab of the Results
    'SVP'       : (10, '%' ),                # tab in the result file
    'Diameter' : (3,  'µm')},                # and assign it to
  'AnotherResult' : 'This is                 # variable resultMap
another result'}

gdrf.saveResults(resultMap,                  # write and save the file
report, "2026")                              # CustomResultFile.gdr
                                             # with report and Result
                                             # map. Result file is
                                             # automatically opened
                                             # in Result Viewer
```



.saveGeometry

Saves the currently loaded structure file to the result folder and adds a geometry map to the result file allowing to load the structure via the **Load Structure** button in the result file and showing a green dot if the structure is loaded.

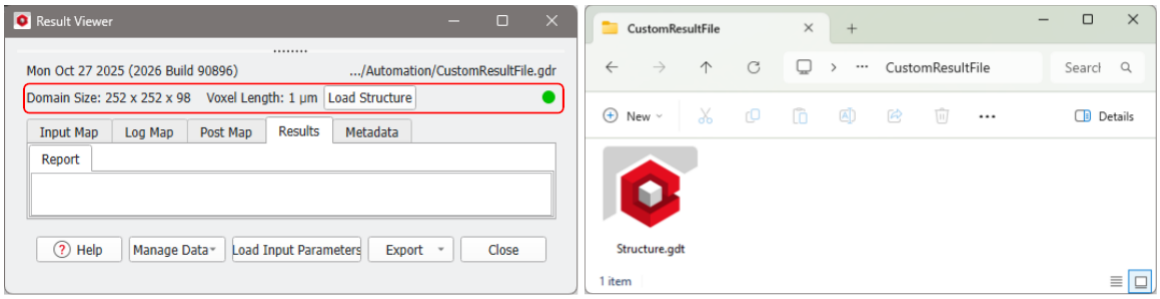
Input:

- gdr file name, string
- GeoDict release, string

Example:

```
from gdr import GDR                          # import gdr
```

```
gdrf =                                  # open a result file
GDR.createEmptyResults("CustomResu      # with the name
ltFile","2026")                         # CustomResultFile.gdr
                                        # and assign it to the
                                        # variable gdrf

gdrf.saveGeometry("2026")               # define text for the
                                        # Report tab and assign
                                        # it to variable report

gdrf.saveResults({},"", "2026")         # write and save the file
                                        # CustomResultFile.gdr
                                        # with empty report and
                                        # empty Result map
```



.setDescription

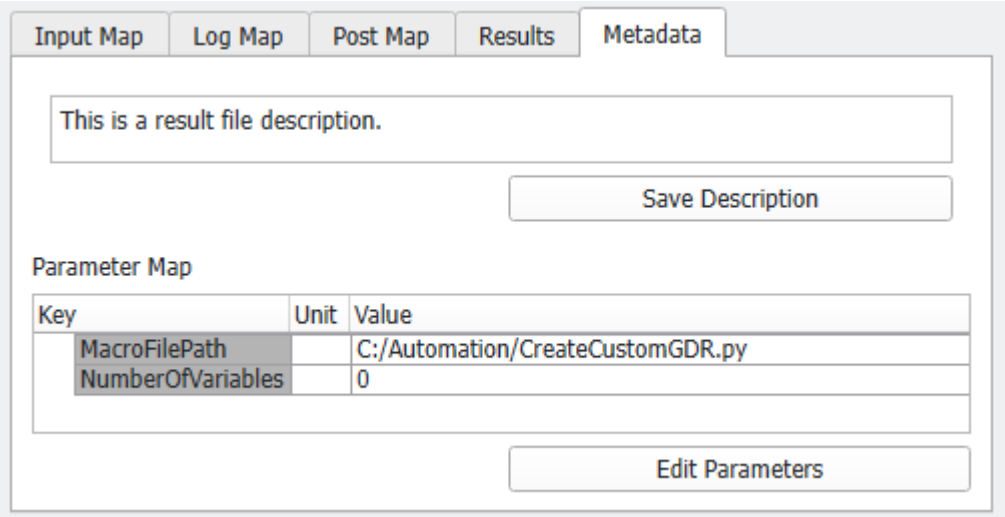
Replaces the default description “Created by macro 'macro file path'.” by a custom description in the **Metadata** tab of the generated result file.

Input:

- description text, string

Example:

```
from gdr import GDR                     # import gdr library

gdrf =                                  # open a result file
GDR.createEmptyResults("CustomResu      # with the name
ltFile","2026")                         # CustomResultFile.gdr

gdrf.setDescription("This is a          # add description to
result file description.")              # Metadata tab

gdrf.saveResults({},"", "2026")         # write and save the file
                                        # CustomResultFile.gdr
                                        # with empty report
                                        # and empty Result map
```



.command = GeoDict command

Adds a custom **Command** name to the generated result file. The command name must be given as a string, and consists of a name for the module, a colon (:), and a name for the solver. The command name can for example be viewed in the **Module** and **Command** columns in the top of the Result Viewer.

Input:

- GeoDict, command, string

Example:

```
from gdr import GDR                    # import gdr library

gdrf =                                 # open a result file
GDR.createEmptyResults("CustomResu     # with the name
ltFile","2026")                        # CustomResultFile.gdr

gdrf.command =                         # add command with module
'ExampleModule:ExampleSolver'          # name ExampleModule and
                                       # solver name
                                       # ExampleSolver
                                       # to result file

gdrf.saveResults({},"", "2026")        # write and save the file
                                       # CustomResultFile.gdr
                                       # with empty report
                                       # and empty Result map
```

	File	Module	Command
1	C:/Automation/CustomResultFile.gdr	ExampleModule	ExampleSolver

Add plots to result file

.addPlot

Adds a plot to the result file with the given plot title, labels, units, values and the graph title. For each new plot added, a new subtab is created in the Plots tab of the Results tab.

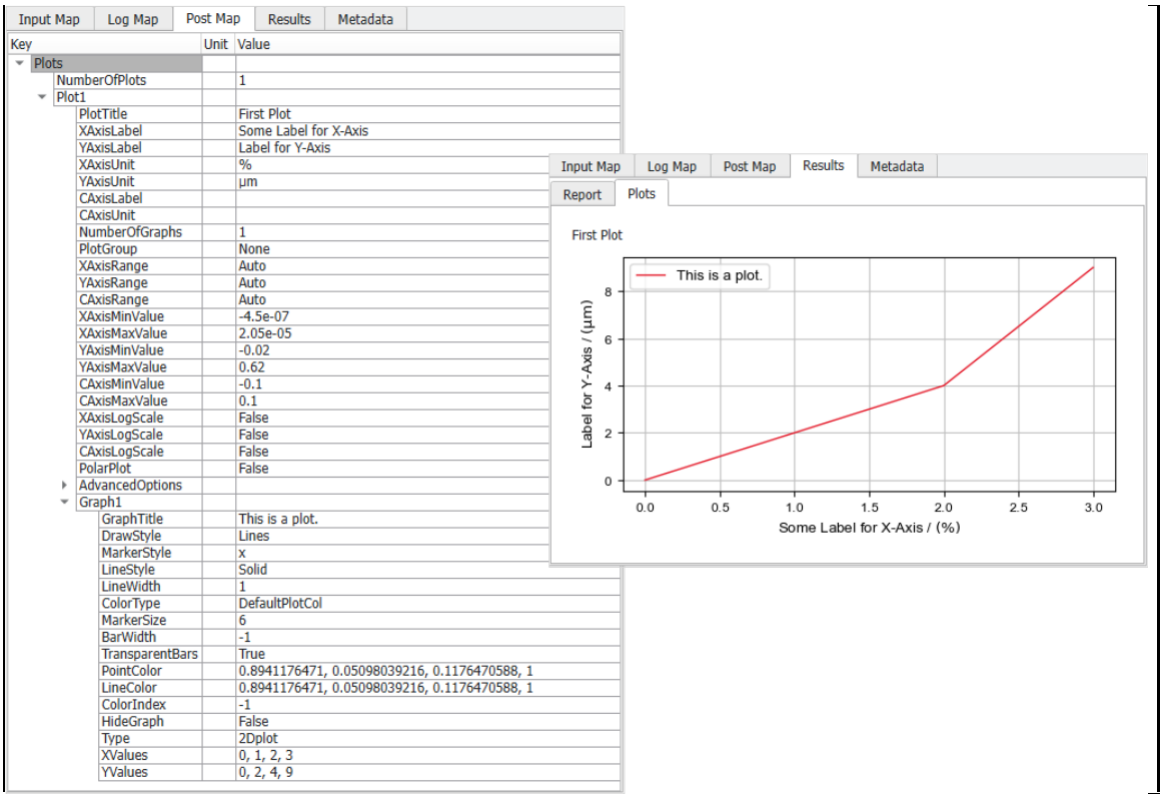
Input:

- plot title, string
- x-axis label, string
- y-axis label, string
- unit for x-values, string
- unit for y-values, string
- x-values, list
- y-values, list
- graph title, string

Example:

```
from gdr import GDR                          # import gdr library

gdrf =                                       # open a result file with
GDR.createEmptyResults("CustomResu           # the name
ltFile","2026")                              # CustomResultFile.gdr
                                             # and assign it to the
                                             # variable gdrf

Plotname  = 'First Plot'                     # define a plot name

XLabel    = 'Some Label for X-               # define X-Axis label
Axis'

YLabel    = 'Label for Y-Axis'               # define Y-Axis label

XValues   = [0,1,2,3]                        # define X-values as a list

YValues   = [0,2,4,9]                        # define Y-values list

GraphName = "This is a plot."                # define a graph name

gdrf.addPlot(Plotname, XLabel,               # add plot to result file
YLabel, '%', 'µm', XValues,                  # with name, labels,
YValues, GraphName)                          # units, values and
                                             # graph name

gdrf.saveResults({},"", "2026")              # write and save the file
```

```
                                        # CustomResultFile.gdr
                                        # with empty report and
                                        # empty Result map
```



.addGraphToPlot

Adds a graph to one of the created plots. Enter the X- and Y- values and the graph title.

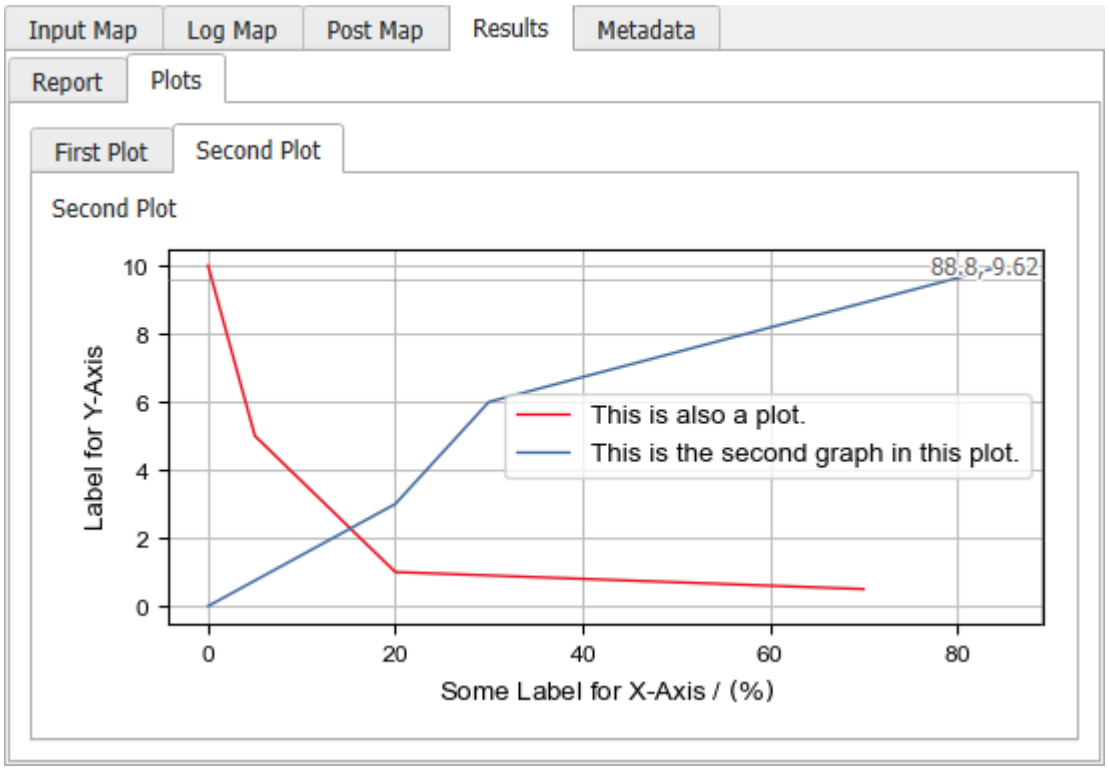
Input:

- plot number, int
- x-values, list
- y-values, list
- graph title, string

Example:

```
from gdr import GDR                     # import gdr library

gdrf =                                  # open a result file with the
GDR.createEmptyResults("CustomResu      # name CustomResultFile.gdr
ltFile","2026")                         # and assign it to the variable
                                        gdrf

Plotname  = 'First Plot'                # define a plot name
```

```
XLabel     = 'Some Label for X-Axis'      # define X-Axis label

YLabel     = 'Label for Y-Axis'           # define Y-Axis label

XValues    = [0,1,2,3]                    # define X-values as a list

YValues    = [0,2,4,9]                    # define Y-values as a list

GraphName = "This is a plot."             # define a graph name

gdrf.addPlot(Plotname, XLabel,            # add plot to result file with
YLabel, '%', 'µm',                        # plotname, labels, units,
   XValues, YValues, GraphName)           # values and graph name

Plotname2  = 'Second Plot'                # define a plot name

XLabel2    = 'Some Label for X-Axis'      # define X-Axis label

YLabel2    = 'Label for Y-Axis'           # define Y-Axis label

XValues2   = [0,5,20,70]                  # define X-values as a list

YValues2   = [10,5,1,0.5]                 # define Y-values list

GraphName2 = "This is also a plot."       # define a graph name

gdrf.addPlot(Plotname2, XLabel2,          # add plot to result file
YLabel2,                                  # with name, labels, units,
  '%', '', XValues2, YValues2,            # values and graph name
GraphName2)

XValues3   = [0,20,30,85]                 # define X-values list

YValues3   = [0,3,6,10]                   # define Y-values list

GraphName3 = "This is the second graph in this plot."   # define a graph name

gdrf.addGraphToPlot(2, XValues3,          # add graph to second plot
YValues3, GraphName3)

gdrf.saveResults({},"", "2026")           # write and save the file
                                          # CustomResultFile.gdr with
                                          # empty report and
                                          # empty result map
```



Add content to report

.addText

Adds text in the **Result – Report** subtab of the generated result file. With this command several lines of text can be added and also text can be placed between tables or images added with the commands addTable and addImage, while the text inserted in the saveResults command always is placed at the end of the report.

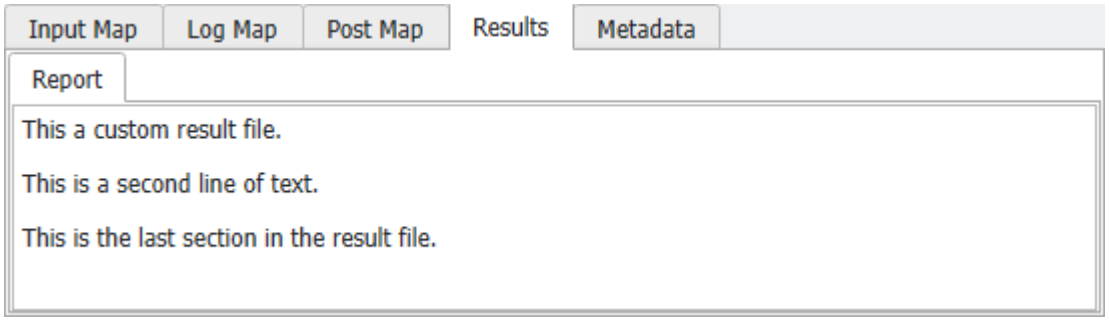
Input:

- text, string

Example:

```
from gdr import GDR                          # import gdr library

gdrf =                                       # open a result file
GDR.createEmptyResults("CustomResu           # with the name
ltFile","2026")                              # CustomResultFile.gdr

gdrf.addText("This a custom result           # add text in Report tab
file.")

gdrf.addText("This is a second               # add more text in
line of text.")                              # Report tab
```

```
report = "This is the last section in the result file."  # define text for the
                                                        # Report tab and assign
                                                        # it to variable report

gdrf.saveResults({}, report, "2026")                    # write and save the file
                                                        # CustomResultFile.gdr
                                                        # with report and an
                                                        # empty Result map
```



.addTitle

Adds titles in the **Result – Report** subtab of the generated result file. This is a similar to the addText command, but with bold font.

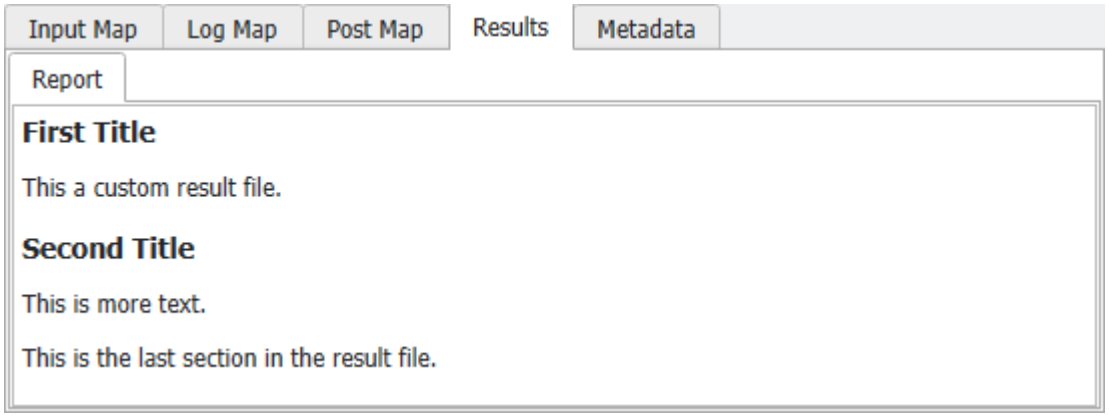
Input:

- title, string

Example:

```
from gdr import GDR                                     # import gdr library

gdrf = GDR.createEmptyResults("CustomResultFile","2026")  # open a result file
                                                          # with the name
                                                          # CustomResultFile.gdr

gdrf.addTitle("First Title")                            # add a title in
                                                        # Report tab

gdrf.addText("This a custom result file.")              # add text
                                                        # in Report tab

gdrf.addTitle("Second Title")                           # add another title
                                                        # in Report tab

gdrf.addText("This is more text.")                      # add more text
                                                        # in Report tab

report = "This is the last section in the result file."  # define text for
                                                         # the Report tab
                                                         # and assign it to
                                                         # variable report
```

```
gdrf.saveResults({}, report,          # write and save the file
"2026")                               # CustomResultFile.gdr
                                      # with report and an
                                      # empty Result map

gd.showGDR('CustomResultFile.gdr')    # open result file
                                      # in Result Viewer
```



.addTable

Adds table to the **Results – Report** subtab of the generated result file.

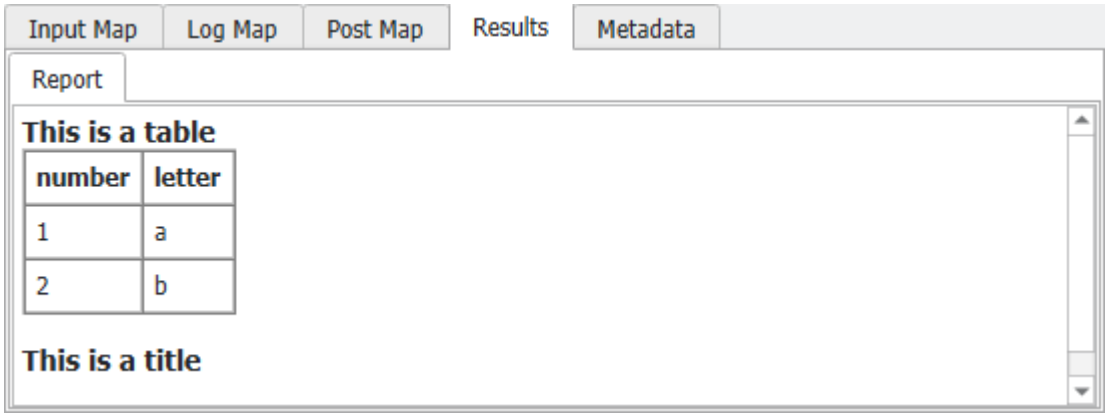
Input:

- title, string
- column headers, list
- table, *list

Example:

```
from gdr import GDR                   # import gdr library

gdrf =                                # open a result file
GDR.createEmptyResults("CustomResu    # with the name
ltFile","2026")                       # CustomResultFile.gdr

col_headers = ["number", "letter"]    # define column headers
                                      # in a list of strings

table = [[1, 2], ["a", "b"]]          # define table as
                                      # list of columns

gdrf.addTable("This is a table",      # add table to
col_headers, *table)                  # Report tab

gdrf.addTitle("This is a title")      # add a title to
                                      # Report tab

report = "This is a report."          # define text for the
```

```
                                    # Report tab and
                                    # assign it to
                                    # variable report

gdrf.saveResults({}, report,        # write and save the file
"2026")                             # CustomResultFile.gdr
                                    # with report and an
                                    # empty Result map
```



.addImage

Adds an image to the **Results – Report** subtab of the generated result file. Additionally, the image is saved to the corresponding result folder.

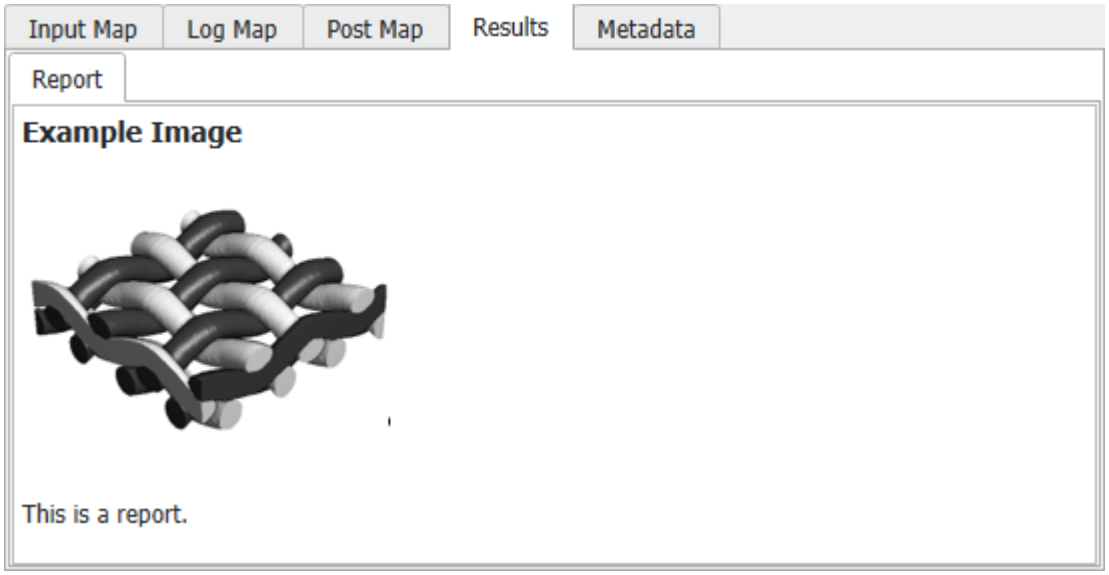
Input:

- image file path, string
- title, string

Example:

```
from gdr import GDR                 # import gdr library

gdrf =                              # open a result file
GDR.createEmptyResults("CustomResu  # with the name
ltFile","2026")                     # CustomResultFile.gdr

gdrf.addImage('../image.png',       # add image to
'Example Image')                    # Report tab

report = "This is a report."        # define text for the
                                    #  Report tab and
                                    # assign it to
                                    # variable report

gdrf.saveResults({}, report,        # write and save
"2026")                             # the file
                                    # CustomResultFile.gdr
                                    # with report and an
```

```
                                        # empty Result map
```



Add content to maps

.inputMap.update

Adds content to the **Input Map** tab of the generated result file. The input map by default already contains the entries ResultFileName and MacroFilePath. The content for the Python dictionary to add can be chosen as desired.

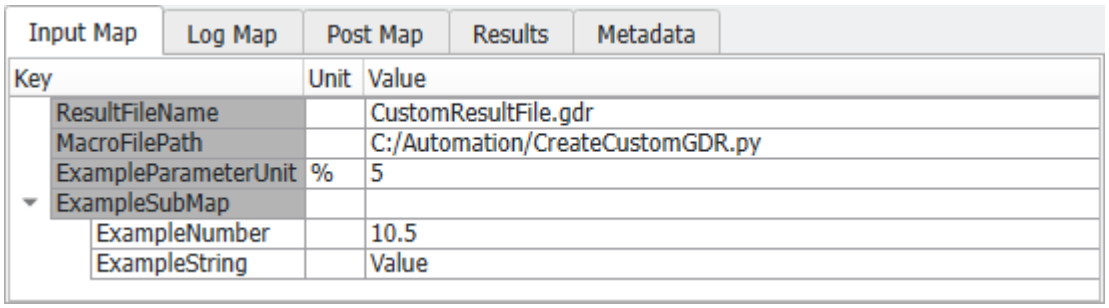
Input:

- input map to add, Python dictionary

Example:

```
from gdr import GDR                     # import gdr library

gdrf =                                  # open a result file
GDR.createEmptyResults("CustomResu      # with the name
ltFile","2026")                         # CustomResultFile.gdr

InputParameters = {                     # assign a Python dictionary
  'ExampleParameterUnit' :(5,'%'),      # containing the input
  'ExampleSubMap' : {                   # parameters as key-value
    'ExampleNumber' : 10.5,             # pairs to variable
    'ExampleString' : 'Value'}}         # InputParameters

gdrf.inputMap.update(InputParamete      # add entries to InputMap
rs)                                     # tab of result file

gdrf.saveResults({},"", "2026")         # write and save the file
                                        # CustomResultFile.gdr with
```

```
                                        # empty report and
                                        # empty Result map
```



.logMap.update

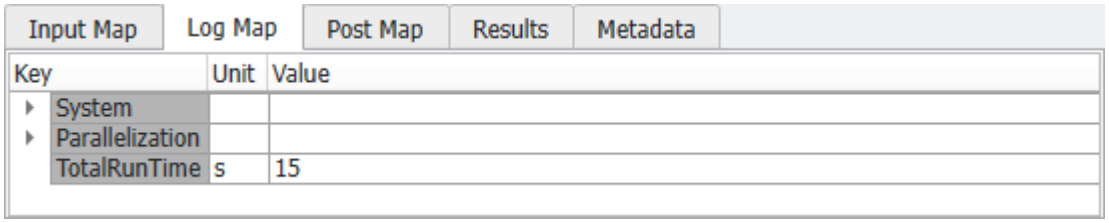
Adds entries to the **Log Map** tab of the generated result file. By default the result file already contains information about the system on which the macro is run.

Input:

- log map to add, Python dictionary

Example:

```
from gdr import GDR                     # import gdr library

gdrf =                                  # open a result file
GDR.createEmptyResults("CustomResu      # with the name
ltFile","2026")                         # CustomResultFile.gdr

LogParameters = {                       # assign a Python dictionary
  'TotalRunTime' :(15,'s')}             # containing the log
                                        # parameters, for example
                                        # the runtime or data about
                                        # the used computer to the
                                        # variable LogParameters

gdrf.logMap.update(LogParameters)       # add the runtime to the
                                        # Log Map tab of the
                                        # result file

gdrf.saveResults({},"", "2026")         # write and save the file
                                        # CustomResultFile.gdr
                                        # with empty report
                                        # and empty Result map
```



.postMap.update

For experienced users, defining all plot parameters manually by adding a **Post Map** creates a **Plot** subtab to the **Results** tab of the generated result file. In most cases, however, it is recommend to add plots via the addPlot and addGraphToPlot[147] commands. In the following example find the keys, that must be given to obtain a plot. For more possible keys refer to **Post Map** tabs in usual GeoDict simulation result files.

Input:

- post processing map, Python dictionary

Example:

```
from gdr import GDR                    # import gdr library

gdrf =                                 # open a result file
GDR.createEmptyResults("CustomResu     # with the name
ltFile","2026")                        # CustomResultFile.gdr

plotParameters = {                     # define plot parameters
  'PlotTitle'       : 'Another         # and assign them to
Plot',                                 # variable plotParameters.
  'XAxisLabel'      : 'X-Axis          # Define labels and units
Label',                                # for both axes.
  'YAxisLabel'      : 'Y-Axis          # Define how the default
Label',                                # axis range should be given.
  'XAxisUnit'       : '',              # Possible values are
  'YAxisUnit'       : '%',             # Automatic, Auto,
  'XAxisRange'      : 'Auto',          # MinMax, and Tight.
  'YAxisRange'      : 'MinMax',        # Define number of graphs
  'YAxisMinValue'   : -5,              # in the plot
  'YAxisMaxValue'   : 15,              # Possible values for
  'NumberOfGraphs' : 1,                # DrawStyle are LinesPoints,
  'Graph1' : {                         # Bars, Lines, Points,
    'GraphTitle'    : 'This is a       # FilledStep, VerticalSpan,
plot',                                 #´and HorizontalSpan.
    'DrawStyle'     : 'Lines',
    'XValues'       : [0,1,2,3],
    'YValues'       : [0,2,4,9]}}


postParameters = {                     # assign Python dictionary
  'Plots' : {                          # containing plot parameters
    'NumberOfPlots' : 1,               # to variable postParameters.
    'Plot1'          :
plotParameters}}

gdrf.postMap.update(postParameters     # add a Post Map tab and
)                                      # plots to result file.

gdrf.saveResults({},"", "2026")        # write and save the file
                                       # CustomResultFile.gdr with
                                       # empty report and
                                       # empty Result map
```

.parameterMap.update

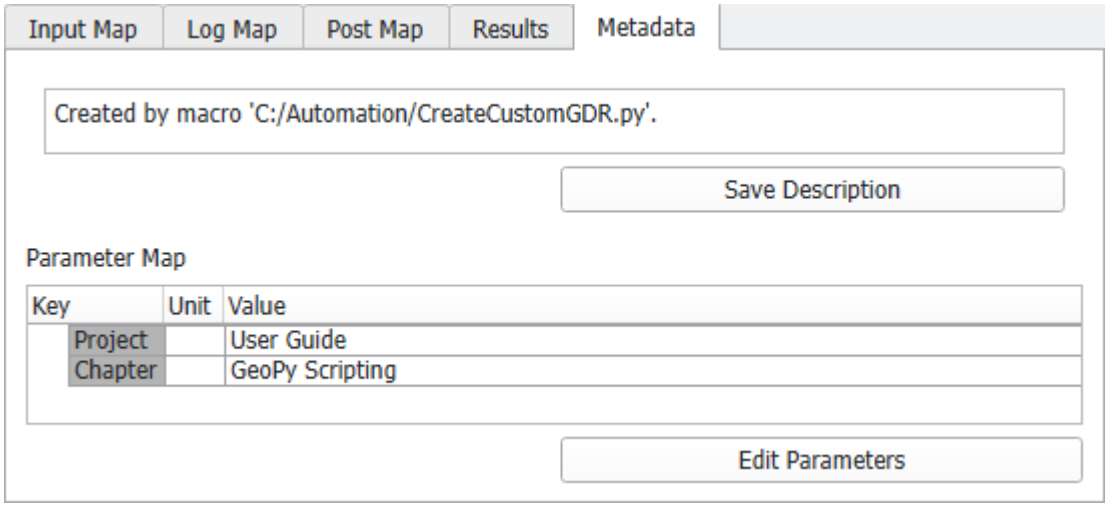
Adds parameters to the **Parameter Map** in the **Metadata** tab of the generated result file. By default, the map already contains the entries MacroFilePath and NumberOfVariables. The content for the Python dictionary can be chosen as desired.

Input:

- parameter map to add, Python dictionary

Example:

```
from gdr import GDR                    # import gdr library

gdrf =                                 # open a result file
GDR.createEmptyResults("CustomResu     # with the name
ltFile","2026")                        # CustomResultFile.gdr

ParameterParameters = {                # assign a Python dictionary
  'Project' : 'User Guide',            # containing parameters to
  'Chapter' : 'GeoPy Scripting'}       # the variable
                                       # ParameterParameters

gdrf.parameterMap =                    # add a Parameter Map
ParameterParameters                    # to result file

gdrf.saveResults({},"", "2026")        # write and save the file
                                       # CustomResultFile.gdr
                                       # with empty report
                                       # and empty Result map
```



.geometryMap.update

For advanced users. Adds explicitly given **Geometry** data to the generated result file. In most cases it is recommended to use the addGeometry 144 command instead to generate the needed data automatically from the currently loaded structure.

The Geometry map must be given correctly. Then, loading the corresponding structure file to GeoDict leads to a green dot in the result viewer. If the structure also is saved to

the result folder, a **Load Structure** button appears in the result file. The geometry Python dictionary must contain the keys shown in the following example. If the structure is in memory, the corresponding values can be contained with GeoPy API functions as shown.

Input:

- geometry map, Python dictionary

Example:

```
from gdr import GDR                    # import

strucHash      =                       # get structure hash
gd.getStructureHash()

strucHash64    =                       # get structure hash 64
gd.getStructureHash64()

strucDesc      =                       # get structure name
gd.getStructureDescription()

nx,ny,nz       =                       # get number of voxels in
gd.getVolDimensions()                  # x-,y- and z-direction

voxelLength    =                       # get voxel length in meter
gd.getVoxelLength()

volDimension   = nx*ny*nz              # compute total number of voxels

svp            =                       # for a structure with two solid
gd.getSolidVolumeFraction()            # materials assigned to material
                                       # ID 1 and ID 2, this computes
                                       # the solid volume percentage

gdrf =                                 # open a result file with the
GDR.createEmptyResults("CustomResu     # name CustomResultFile.gdr
ltFile","2026")

GeometryParameters = {                 # assign a Python dictionary
  'Hash'                 :             # containing geometry data
strucHash,                             # to the variable
  'Hash64'               :             # GeometryParameters
strucHash64,
  'FileName'             :
'Example.gdt',
  'NX'                   : nx,
  'NY'                   : ny,
  'NZ'                   : nz,
  'UseBoxels'            : False,
  'VoxelLength'          :
(voxelLength,'m'),
  'SolidVolumeFraction'  : svp}
```

```
gdrf.geometryMap.update(GeometryPa  # add geometry data to
rameters)                           # result file

gdrf.saveResults({},"", "2026")     # write and save the file
                                    # CustomResultFile.gdr with
                                    # empty report and
                                    # empty Result map
```

Note! When using .update on a map, the current content of the map is kept and the content of your custom map is added below. If you want to replace it with your custom map, instead of `.update(newmap)` use `= newmap`.

1.7.6 Access to 3D Result Files

The **GeoDict Universal File** (**GUF**) format is a generic file format that contains large amounts of data that were computed with GeoDict. Most structures and result fields in GeoDict are GUF files, e.g., *.gdt, *.vap, *.gpp, ... Using binary data avoids a loss of precision and provides efficient read and write operations.

GUF files begin with a header in text format, which (for small GUF files) can be inspected by opening the file with a text editor. The header is followed by binary data. Meta data describing the binary data is contained in the header and is line-based with pairs of key and value per line.

GeoDict provides a GUF python library in GeoPy to access GUF files without loading them to GeoDict.

Learn about GUF files:

- Structure of a GUF File 159
  Learn about the structure of different GUF files.
- Access GUF Files 163
  Here, the functions to access GUF files are described.

Structure of a GUF File

Every GUF file consists of two sections: The **Header section** and the **Binary Data section**.

The **Header section** gives information about the binary content in the **Binary Data section** in form of key - value pairs, similar to a Python dictionary. The meta information is stored in humanly readable ASCII and has (at least) 256 bytes. However, it must not be edited, as the header must correspond to the binary data.

The header consists of several blocks and always starts with the GUF version consisting of the file format and its version. The example below is GDT3, i.e., a version 3 *.gdt file. This is the default *.gdt file format for GeoDict structure files since GeoDict 2023.

```
1  GufVersion GDT3
2  HeaderLength 4608
3  DateOfCreation Mon Oct 27 2025 17:24:02
4  GeoDictEdition 2026
5  GeoDictRevision 90896
6  GeoDictPlatform 64 bit Windows
7  VoxelLength 1e-06
8  GADMatchesVG 1
9  PeriodicX 0
10 PeriodicY 0
11 PeriodicZ 0
12 OriginX 0
13 OriginY 0
14 OriginZ 0
15 Description Geometry created with FiberGeo
16 StructureHash64 13529921876377552316
17 Nx 100
18 Ny 100
19 Nz 100
20 NumberOfImages 1
21 EntriesOfImages 1
22 NamesOfImages Structure
23 Image1:Names Voxels
24 Image1:Order position
25 Image1:Grids center
26 Image1:Meaning indexed
27 Image1:Types uint8
28 Image1:Units 1
29 Image1:Compression rle
30 Image1:Offset 4608
31 Image1:Length 59042
32 NumberOfMaps 4
33 NamesOfMaps GAD,GADStats,Materials,MaterialDatabase
34 Map1:Compression zlib
35 Map1:Offset 63650
36 Map1:Length 4265
37 Map2:Compression zlib
38 Map2:Offset 67915
39 Map2:Length 231
40 Map3:Compression zlib
41 Map3:Offset 68146
42 Map3:Length 1058
43 Map4:Compression zlib
44 Map4:Offset 69204
45 Map4:Length 2110
```

Creation Data

Image Data

File Specific Data

Creation Data

The **Creation Data** block provides information about the creation of the file, e.g., the creation time and the used GeoDict revision.

```
1  GufVersion GDT3
2  HeaderLength 4608
3  DateOfCreation Mon Oct 27 2025 17:24:02
4  GeoDictEdition 2026
5  GeoDictRevision 90896
6  GeoDictPlatform 64 bit Windows
```

Image Data

Detailed information about the **Image Data** is given afterwards. Image Data is stored in a sequence of images as fields.

A full image has Nx by Ny by Nz entries, corresponding to the domain size of the structure / result field in voxels.

The example file has 100 voxels in X-, Y- and Z-direction and one image with the name Structure.

```
VoxelLength 1e-06
GADMatchesVG 1
PeriodicX 0
PeriodicY 0
PeriodicZ 0
OriginX 0
OriginY 0
OriginZ 0
Description Geometry created with FiberGeo
StructureHash64 13529921876377552316
Nx 100
Ny 100
Nz 100
NumberOfImages 1
EntriesOfImages 1
NamesOfImages Structure
Image1:Names Voxels
Image1:Order position
Image1:Grids center
Image1:Meaning indexed
Image1:Types uint8
Image1:Units 1
Image1:Compression rle
Image1:Offset 4608
Image1:Length 59042
```

File Specific Data

At the end of the header, **File Specific Data** blocks can be found, e.g., map data, info data, and array data.

Additional data in *.gdt files is described by stringmaps, that are maps consisting of key-value pairs, similar to a Python dictionary. Thus, the specific block in the example contains map information.

```
NumberOfMaps 4
NamesOfMaps GAD,GADStats,Materials,MaterialDatabase
Map1:Compression zlib
Map1:Offset 63650
Map1:Length 4265
Map2:Compression zlib
Map2:Offset 67915
Map2:Length 231
Map3:Compression zlib
Map3:Offset 68146
Map3:Length 1058
Map4:Compression zlib
Map4:Offset 69204
Map4:Length 2110
```

Binary Data

In the example file, there are four maps with the names GAD, GADStats, Materials, and MaterialDatabase. The **Binary Data** section of the example file is shown here.

Volume Field Data

In a second example, a flow simulation was run on the structure. The GUF file FlowField_z.vap file is produced and shown here.

The flow was computed in Z-direction. The file contains two images with the names Velocity and Pressure. The velocity image contains three fields and the pressure image one. The velocity fields are called VelocityX, VelocityY and VelocityZ.

```
GufVersion VAP1
HeaderLength 1024
DateOfCreation 27.10.25  17:25:40
GeoDictEdition 2026
GeoDictRevision 90896
GeoDictPlatform 64 bit Windows
BoundaryConditionX Periodic
BoundaryConditionY Periodic
BoundaryConditionZ Periodic
EntriesOfImages 3,1
Experiment PressureDrop
FlowDirection BottomToTopZ
Image1:Grids left,front,bottom
Image1:Meaning vector
Image1:Names VelocityX,VelocityY,VelocityZ
Image1:Types float,float,float
Image1:Units m/s,m/s,m/s
Image2:Grids center
Image2:Names Pressure
Image2:Types float
Image2:Units Pa
InletLengthX 0
InletLengthY 0
InletLengthZ 10
MeanVelocityOutput 0.0003447258673
NameOfCreator EJStokes
NamesOfImages Velocity,Pressure
NumberOfImages 2
Nx 100
Ny 100
Nz 100
OutletLengthX 0
OutletLengthY 0
OutletLengthZ 10
PressureDropInput 0.02
VoxelLength 1e-06,1e-06,1e-06
```

Particle Specific Data

Result files generated by the particle tracker in FilterDict and AddiDict (*.gpp) contain a large information block providing details about the simulation.

The particle positions are described by arrays. The example file contains one array with 12 columns and 3600 rows.

```
GufVersion GPP3
HeaderLength 13312
DateOfCreation Mon Oct 27 2025 17:33:37
GeoDictEdition 2026
GeoDictRevision 90896
GeoDictPlatform 64 bit Windows
Info:Charge NONE
Info:CollisionDiameter NONE
Info:Density CONSTANT
Info:DensityValue 2650
Info:DepositionDiameter NONE
```

```
NumberOfArrays 1
NamesOfArrays ParticlePositions
Array1:NumberOfColumns 12
Array1:NumberOfRows 3627
Array1:ColumnNames ID,Type,Position X,Position Y,Position Z,Velocity X,Velocity Y,Velocity Z,Time,Collision Count,Status,Multiplicity
Array1:Types int64,int32,double,double,double,double,double,double,double,int32,int8,int32
Array1:Units 1,1,m,m,m,m/s,m/s,m/s,s,1,1,1
Array1:Offset 13312
Array1:Length 279279
Info:TotalMultiplicitySum 3627
```

Access GUF Files

The GeoPy library provides read-only access for GUF files, using the keys and values from the header. To use this library in the top of the Python file import the library with the following command:

```
from guf import GUF
```

Then, access the desired file and store it in a variable, e.g., guf_file. Therefore, insert the file path of the desired file in the parenthesis of the function **GUF** as follows:

```
guf_file = GUF("example.vap")
```

There are four functions for GUF files described in the following, accessing header, images, arrays and maps.

.getHeader

This function returns the complete header as a stringmap. The values contained in this stringmap can be accessed by adding the corresponding keys in square brackets.

Input: None

Example:

```
from guf import GUF                    # import GUF library

guf_file =                             # access GUF file
GUF("StokesResult/FlowField_z.vap"     # FlowField_z.vap
)

guf_header = guf_file.getHeader()      # assign the header
                                       # stringmap to the
                                       # variable guf_header

print(guf_header)                      # print the complete
                                       # header to the
                                       # GeoDict console

imagenumber =                          # assign the number of
guf_header["NumberOfImages"]           # images to the
                                       # variable imagenumber

gd.msgBox(f"The file contains          # show message dialog
{imagenumber} images.")
```

.getImage

This function returns the specified image as numpy array. Enter the image name inside the parenthesis as a string. Find the image names in the header. To access a volume field from the image, enter the corresponding field name in square brackets.

```
26  NameOfCreator EJStokes
27  NamesOfImages Velocity,Pressure
28  NumberOfImages 2
```

```
14  Image1:Meaning vector
15  Image1:Names VelocityX,VelocityY,VelocityZ
16  Image1:Types float,float,float
```

Basically, this function does the same as the gd.getVolumeField() 81 function described in the general API functions 66, but here no volume field needs to be loaded in GeoDict.

Input:

- image name, string

Note: The getImage function is not recommended for compressed images, as currently the function cannot decompress the image and returns only an 1-dimensional array. Thus, the fields cannot be accessed. For compressed images, the key Image#:Compression can be found in the header. Thus, for these images it is recommended to use the **gd.getStructure** 73 or the **gd.getVolumeField** 80 functions.

```
28  Image1:Units 1
29  Image1:Compression rle
30  Image1:Offset 4608
```

```
27  Image1:Grids                left,front,bottom
28  Image1:Compression          LIR-Tree
29  Image1:CompArgs:Structure   Tree
```

Example:

```
from guf import GUF                    # import GUF library

guf_file =                             # access GUF file
GUF("StokesResult/FlowField_z.vap"     # FlowField_z.vap
)

guf_image =                            # assign the numpy array
guf_file.getImage("Velocity")          # corresponding to the
                                       # image Velocity to the
                                       # variable guf_image

guf_field = guf_image["VelocityX"]     # assign the numpy array
                                       # corresponding to the
                                       # flow field VelocityX to
                                       # the variable guf_field

gd.msgBox(f"The velocity at            # show a message dialog
position (50,50,50) in the             # for the velocity at
Velocity X field is {guf_field[50]     # position (50,50,50)
[50][50]}.")
```

.getArray

This function returns the specified array as a numpy array. Enter the array name inside the braces as a string. Find the array names in the header. For a single column add the corresponding column name in square brackets.

```
253   NumberOfArrays 1
254   NamesOfArrays ParticlePositions
255   Array1:NumberOfColumns 12
256   Array1:NumberOfRows 3627
257   Array1:ColumnNames ID,Type,Position X,Position Y,Position Z,Velocity X,Velocity
      Y,Velocity Z,Time,Collision Count,Status,Multiplicity
258   Array1:Types
      int64,int32,double,double,double,double,double,double,double,int32,int8,int32
259   Array1:Units 1,1,m,m,m,m/s,m/s,m/s,s,1,1,1
```

This function only works, if the GUF file contains arrays (e.g., FilterDict *.gpp files). There are many very helpful **FilterDict Particle specific Functions**, but for the getArray function the trajectories do not need to be loaded in GeoDict.

Input:

- array name, string

Example:

```
from guf import GUF                     # import GUF library

guf_file =                              # access FilterDict result
GUF("FilterLifeTime/Batch00001/Tra      # file TrackerFinalParticles.gpp
ckerFinalParticles.gpp")

guf_array =                             # assign the numpy array
guf_file.getArray("ParticlePositio      # containing the particle
ns")                                    # positions to the variable
                                        # guf_array

id_5  = guf_array["ID"][5]              # assign fifth element in
                                        # the column ID to the
                                        # variable id_5
                                        # (count starts with 0)

pos_5 = guf_array["Position X"][5]      # assign fifth element in
                                        # the column Position X
                                        # to the variable pos_5
                                        # (count starts with 0)

time_5 = guf_array["Time"][5]           # assign fifth element
                                        # in the column Time to
                                        # the variable time_5
                                        # (count starts with 0)

gd.msgBox(f"The particle with ID        # show message dialog
{id_5} has the X-position {pos_5}
at time {time_5}.")

guf_row = guf_array[5]                  # assign the numpy array
                                        # containing the sixth
                                        # entry of all columns
                                        #  to the variable guf_row
```

```
gd.msgBox(f"The particle with ID      # show the same message
{guf_row[0]} has the X-position       # dialog as before
{guf_row[2]} at time
{guf_row[8]}.")
```

.getMap

This function returns the specified map as a stringmap, consisting of key – value pairs. Enter the stringmap name inside the braces as a string. Find the map names in the header. This function only works for *.gdt files.

```
32   NumberOfMaps 4
33   NamesOfMaps GAD,GADStats,Materials,MaterialDatabase
```

There are many very helpful API functions[66] applicable for structure files (e.g., gd.getGADObject[78]), but for the getMap function the structure does not need to be loaded in GeoDict.

To access only the desired information of the stringmap add the corresponding keys in square brackets. The needed keys can be found out by printing the desired map in the GeoDict console.

Input:

- map name, string

Example:

In the example below, the GAD statistics map is printed to the console and the number of objects in the 14th Z-slice is returned in a message dialog.

```
from guf import GUF                    # import GUF library

guf_file =                             # access GUF file Structure.gdt
GUF("FiberGeo/Structure.gdt")          # in the folder FiberGeo

guf_map =                              # assign the stringmap of the
guf_file.getMap("GADStats")            # GAD statistics for all 2D
                                       # slices to the variable guf_map

print(guf_map)                         # print the stringmap to the
                                       # GeoDict console

objectscount_Z =                       # assign the string containing
guf_map["PerSliceObjectCountsZ"]       # statistics for the Z-slices to
                                       # the variable objectscount_Z

objectscount_Z =                       # split the string by commas,
objectscount_Z.split(',')              # and obtain a list

count_Z_slice_13 =                     # assign the 14th entry in the
objectscount_Z[13]                     # list (index 13 as counting
```

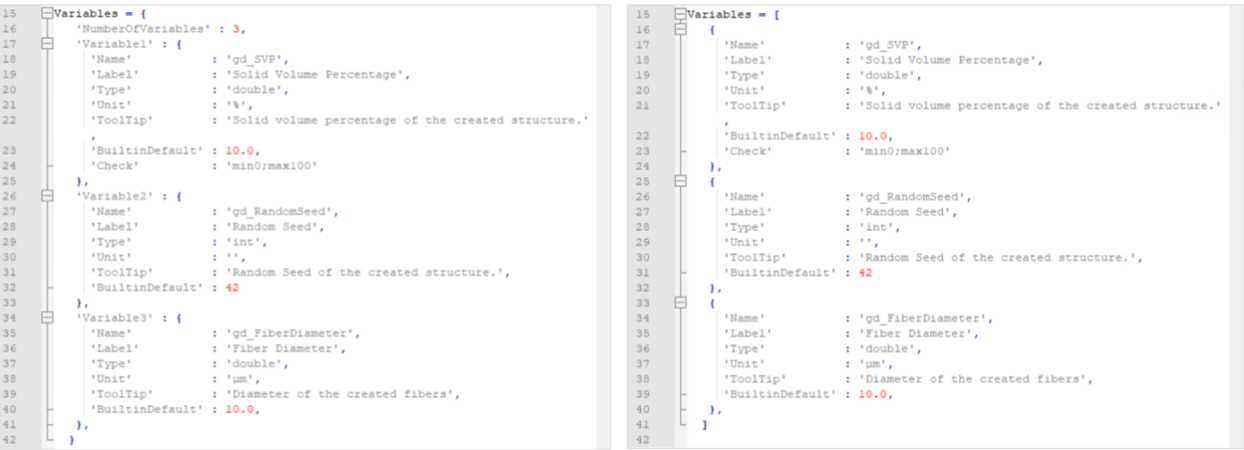
```
                                    # starts with 0) to the variable
                                    # count_Z_slice

gd.msgBox(f" In Z-slice 14 there    # show a message dialog.
are {count_Z_slice_13} objects.")
```

1.7.7 Error Reporting

Learn about common error messages to debug your Python scripts. Exceptions which happen in Python code and are not caught in Python code (e.g., when you try to open a file that does not exist) trigger an error dialog box in GeoDict and terminate the execution of the macro.

In the following find error messages and their explanations for common errors.

Variables Dictionary



Note! Since GeoDict 2025 the variables section does not need to be a dictionary anymore, but instead can be a list of dictionaries. Thus, the errors regarding variable numbering also can be easily solved by changing the section to the new syntax. The old syntax, however, is still supported.

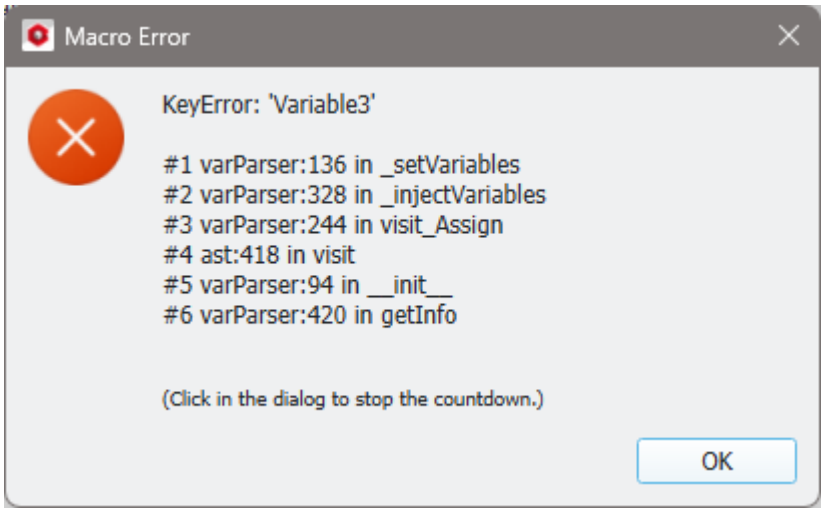
```
Variables = {
  'NumberOfVariables' : 3,
  'Variable1' : {
    'Name'           : 'gd_SVP',
    'Label'          : 'Solid Volume Percentage',
    'Type'           : 'double',
    'Unit'           : '%',
    'ToolTip'        : 'Solid volume percentage of the created structure.'
    ,
    'BuiltinDefault' : 10.0,
    'Check'          : 'min0;max100'
  },
  'Variable2' : {
    'Name'           : 'gd_RandomSeed',
    'Label'          : 'Random Seed',
    'Type'           : 'int',
    'Unit'           : '',
    'ToolTip'        : 'Random Seed of the created structure.',
    'BuiltinDefault' : 42
  },
  'Variable3' : {
    'Name'           : 'gd_FiberDiameter',
    'Label'          : 'Fiber Diameter',
    'Type'           : 'double',
    'Unit'           : 'µm',
    'ToolTip'        : 'Diameter of the created fibers',
    'BuiltinDefault' : 10.0,
  },
}
```

Variables block - old syntax

```
Variables = [
  {
    'Name'           : 'gd_SVP',
    'Label'          : 'Solid Volume Percentage',
    'Type'           : 'double',
    'Unit'           : '%',
    'ToolTip'        : 'Solid volume percentage of the created structure.'
    ,
    'BuiltinDefault' : 10.0,
    'Check'          : 'min0;max100'
  },
  {
    'Name'           : 'gd_RandomSeed',
    'Label'          : 'Random Seed',
    'Type'           : 'int',
    'Unit'           : '',
    'ToolTip'        : 'Random Seed of the created structure.',
    'BuiltinDefault' : 42
  },
  {
    'Name'           : 'gd_FiberDiameter',
    'Label'          : 'Fiber Diameter',
    'Type'           : 'double',
    'Unit'           : 'µm',
    'ToolTip'        : 'Diameter of the created fibers',
    'BuiltinDefault' : 10.0,
  },
]
```

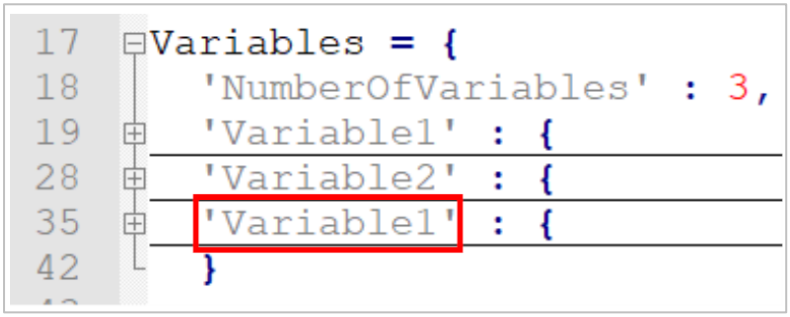
Variables block - new syntax

KeyError: 'Variable#'

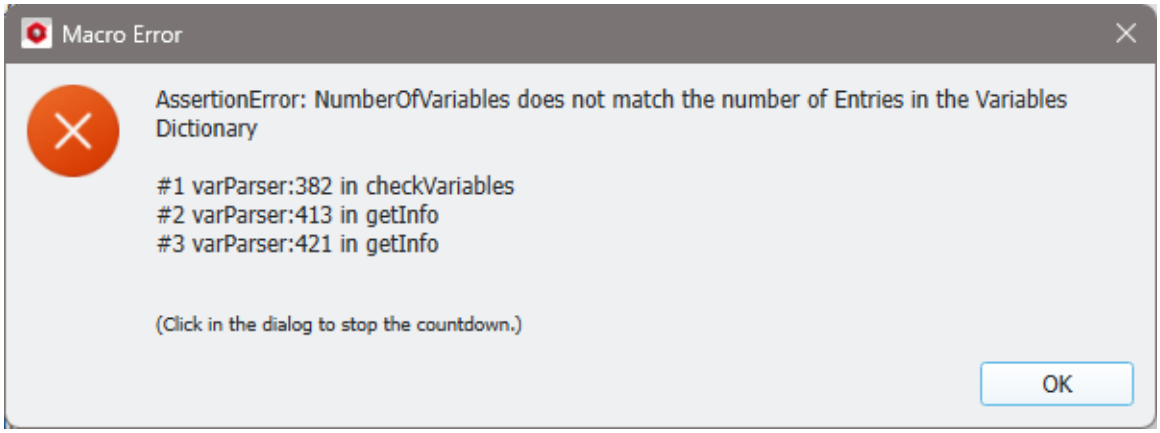


There are not as many variables given in the dictionary as given by the NumberOfVariables entry in the Variables dictionary.

A common error leading to this message can also be, that the third variable was not named Variable3 after copying and pasting Variable1 for example.



AssertionError: NumberOfVariables does not match the number of entries in the Variables Dictionary

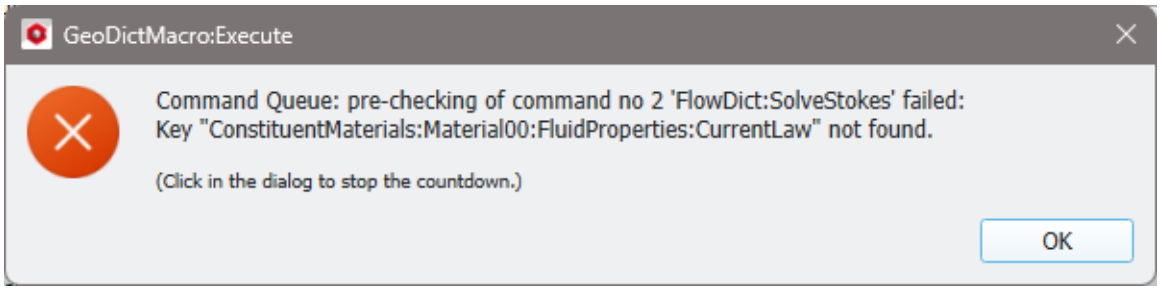


This error happens if there are more variable entries than the NumberOfVariables parameter determines.

```
17  Variables = {
18    'NumberOfVariables' : 2,
19    'Variable1' : {
28    'Variable2' : {
35    'Variable3' : {
42    }
```

GeoDict Commands

Command Queue: pre-checking of command no # ... failed: Error while reading settings and materials for ...



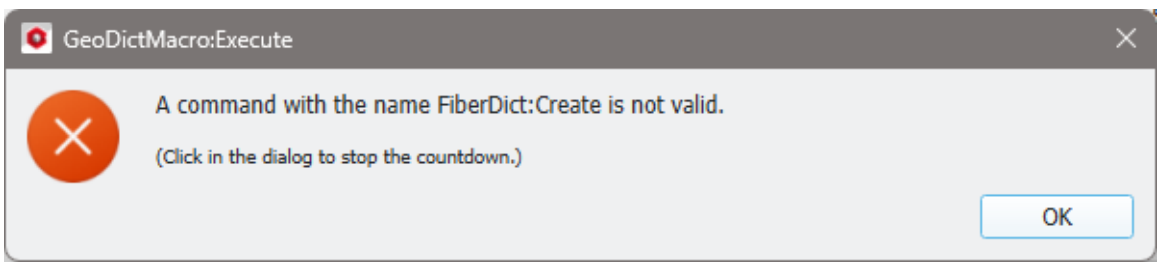
This error message means that needed keys in the corresponding GeoDict command dictionary are missing. This can happen if commands from a macro recorded with an earlier GeoDict release are copied into a newer macro, because the command parameters may have changed. This issue can easily be resolved by explicitly giving the right release year as input argument for the gd.runCmd function, as by default the release is taken from the macro file header.

```
1  Header = {
2    'Release'      : '2026',
3    }
```

```
174  gd.runCmd("FlowDict:SolveStokes", SolveStokes_args_1, Header['Release'])
```

```
174  gd.runCmd("FlowDict:SolveStokes", SolveStokes_args_1, '2022')
```

A command with the name X:Y is not valid.



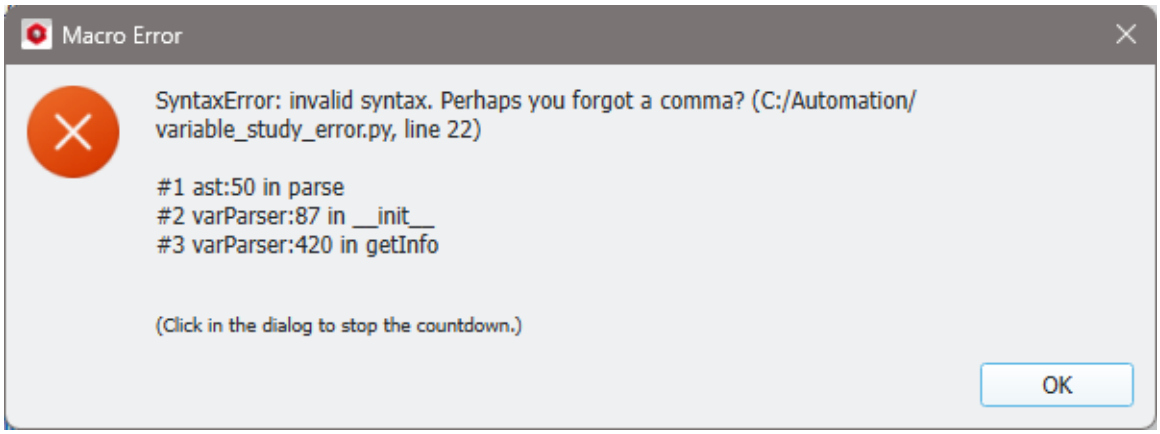
If the given command name does not exist, for example when typing it manually or changing it after recording, this message appears. The command in this example Fiber**Dict**:Create should be Fiber**Geo**:Create.

```
gd.runCmd("FiberDict:Create", Create_args_1, Header['Release'])
gd.runCmd("FiberGeo:Create", Create_args_1, Header['Release'])
```

Invalid Syntax

There are many possibilities to obtain a syntax error. Some of the most common syntax errors are:

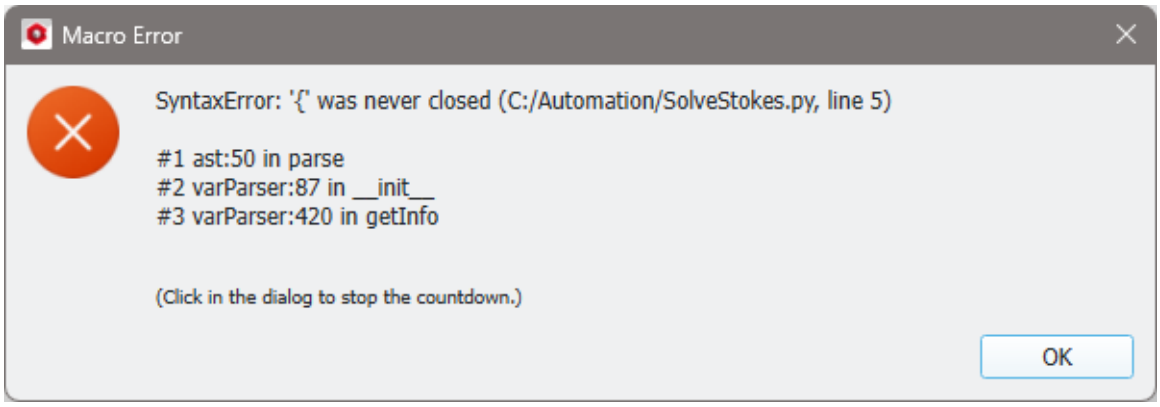
SyntaxError: invalid syntax. Perhaps you forgot a comma? (macrofilepath, line)



This means, a comma in the end of a line in a Python dictionary is missing.

```
15  Variables = [
16      {
17          'Name'           : 'gd_SVP',
18          'Label'          : 'Solid Volume Percentage',
19          'Type'           : 'double',
20          'Unit'           : '%',
21          'ToolTip'        : 'Solid volume percentage of the created structure.'
            ,
22          'BuiltinDefault' : 10.0
23          'Check'          : 'min0;max100'
24      },
```

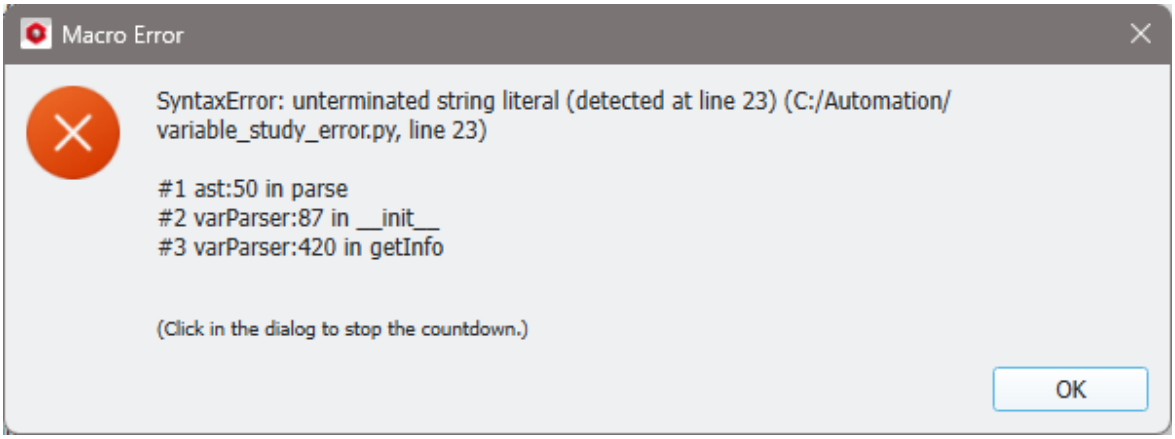
SyntaxError: invalid syntax. '{' was never closed (macrofilepath, line)



This, means, a closing bracket is missing somewhere.

```
  5 SolveStokes_args_1 = {
  6   'ResultFileName'        : 'StokesResult.gdr',
  7   'Experiment' : {
 33   'SolverData' : {
109   'Grid' : {
115   'ConstituentMaterials' : {
116     'Temperature' : (293.15, 'K'),
117     'Material00' : {
118       'Type'             : 'Fluid',
119       'Name'             : 'Air',
120       'Information'      : '',
121       'FluidProperties' : {
122         'CurrentLaw'    : 1,
123         'NumberOfLaws' : 1,
124         'MaterialLaw1' : {
125           'Name'       : 'Material Law 1',
126           'Density'    : (1.204, 'kg/m^3'),
127           'Viscosity' : (1.834e-05, 'kg/(ms)'),
128
129         },
130       },
131     'Material01' : {
```

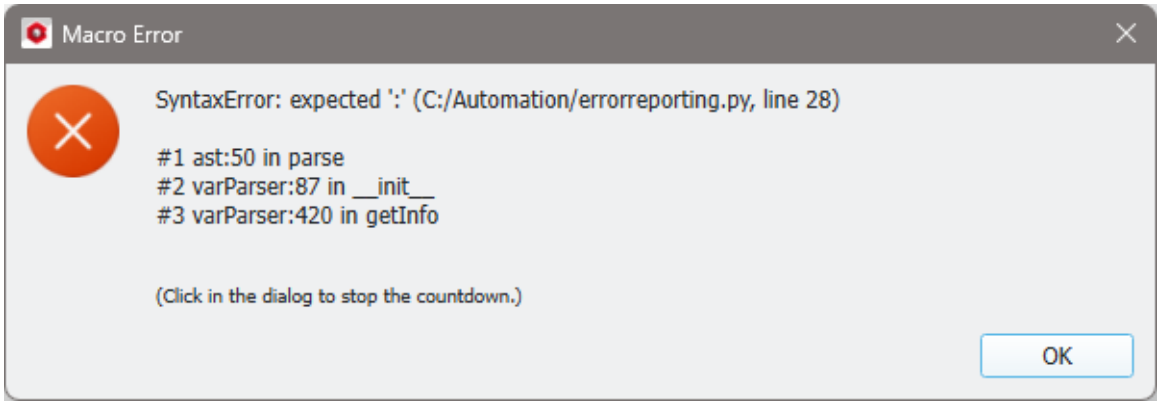
SyntaxError: invalid syntax. unterminated string literal (macrofilepath, line)



This means, a quotation ending is missing somewhere.

```
15  Variables = [
16      {
17          'Name'           : 'gd_SVP',
18          'Label'          : 'Solid Volume Percentage',
19          'Type'           : 'double',
20          'Unit'           : '%',
21          'ToolTip'        : 'Solid volume percentage of the created structure.'
            ,
22          'BuiltinDefault' : 10.0,
23          'Check'          : 'min0;max100
24      },
```

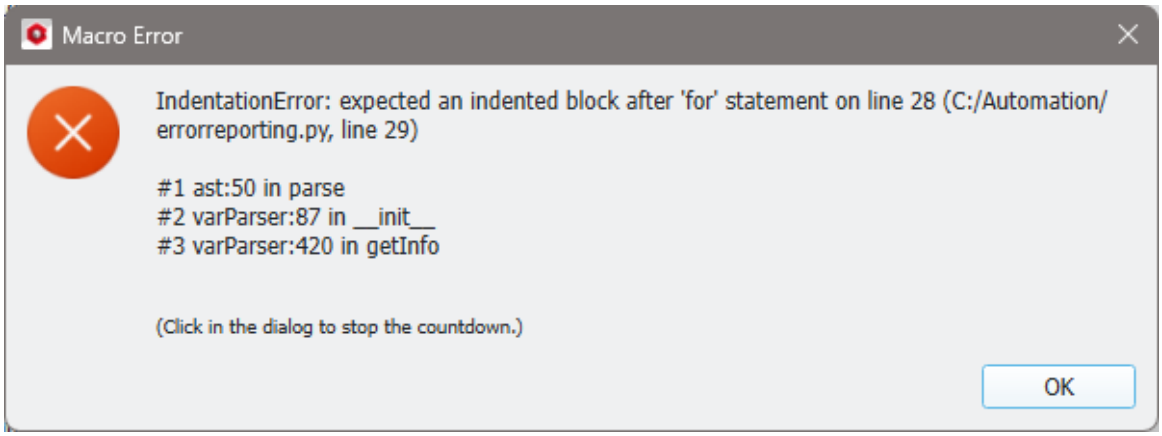
SyntaxError: invalid syntax. expected ':' (macrofilepath, line)



This means there is a colon missing (e.g., in definitions or in the line defining a loop.

```
28  for i in range(5)
29      print(i)
```

IndentationError: expected an indented block (macrofilepath, line)



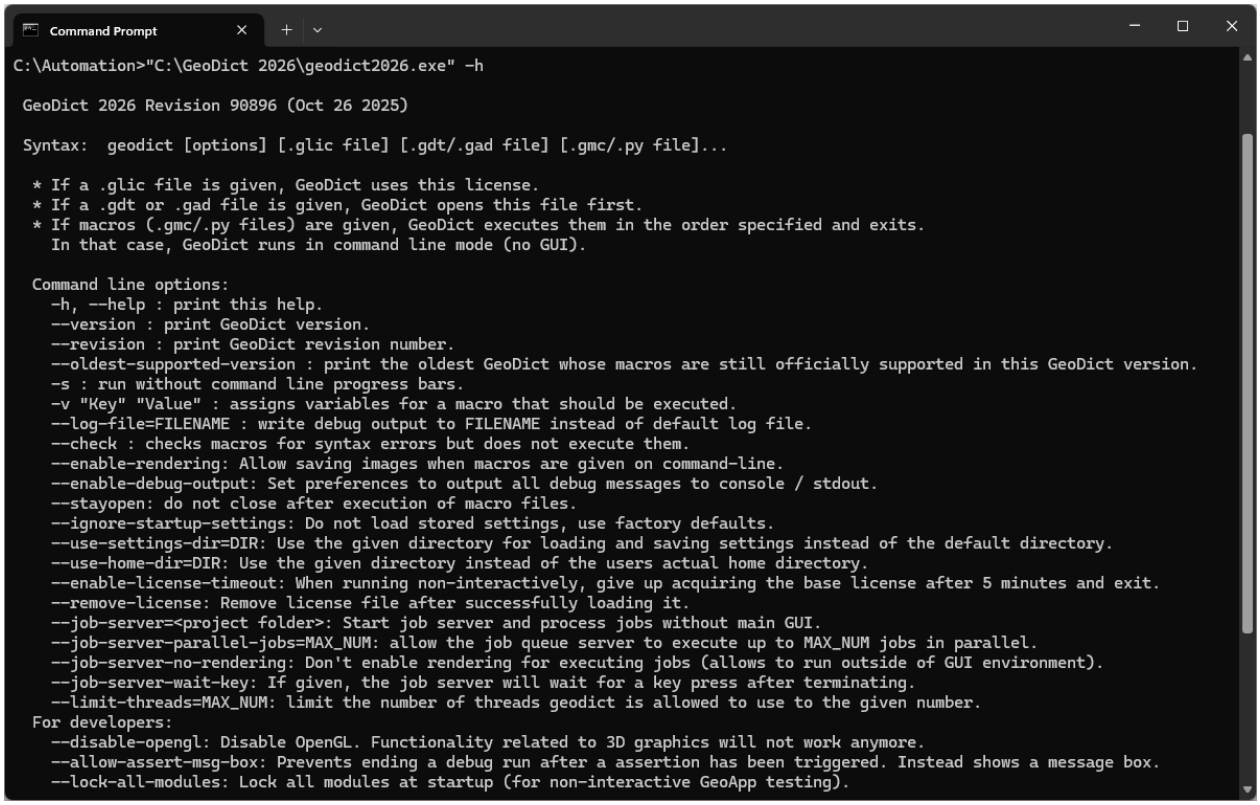
This error message appears, if no indented block is found, where it is expected, e.g., in a loop ('for' statement).

```
28  for i in range(5):
29  print(i)
```

1.8 Running from Command Line

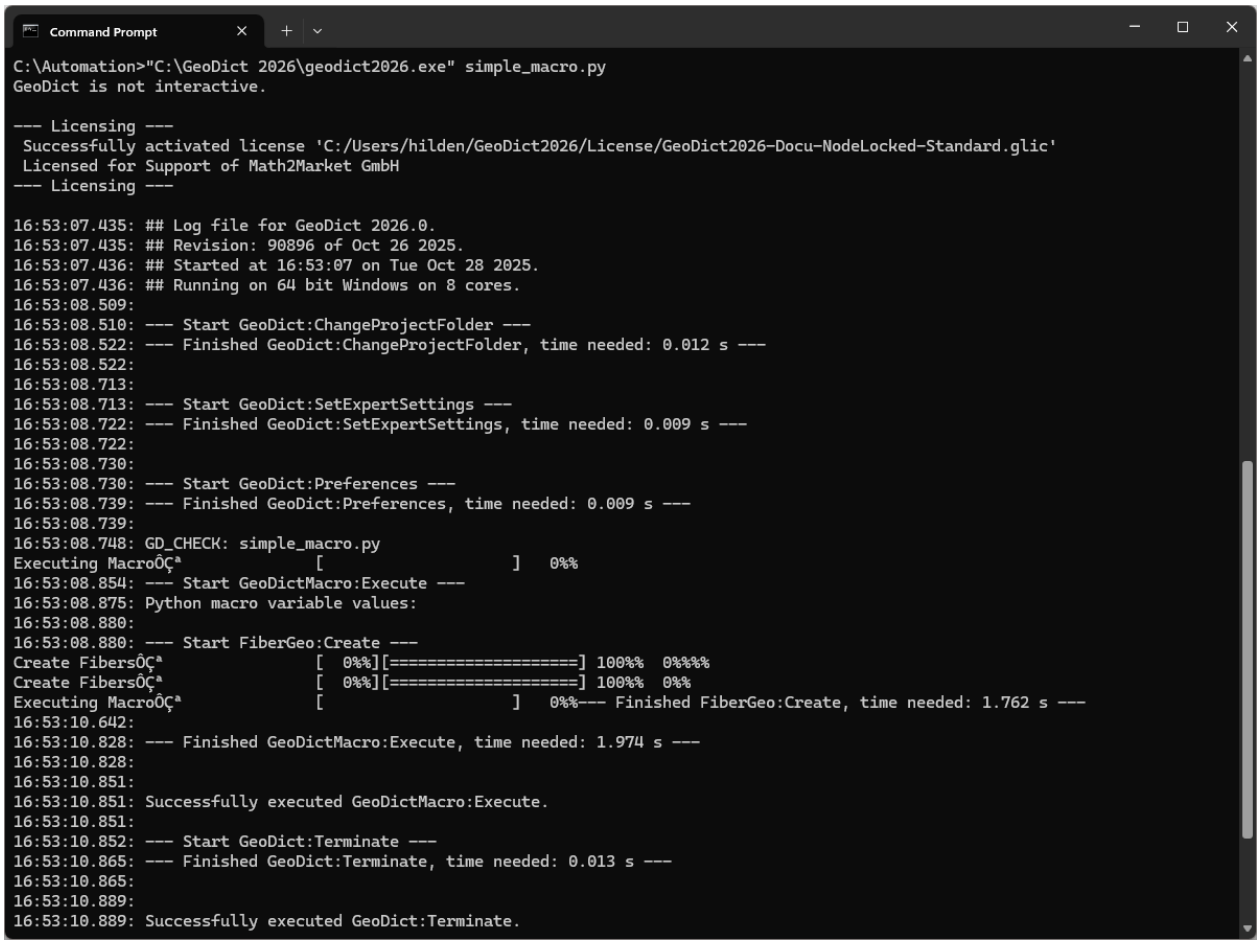
Being comfortable with the command prompt, it is a fast possibility to run GeoDict from the command line without the GUI. Although it is possible to open GeoDict from the command line (**>>Installationpath\GeoDict 2026\geodict2026.exe**), it is not necessary for running macros. The following command prints a helpful list of commands:

>>"Installation-path\geodict2026.exe" -h



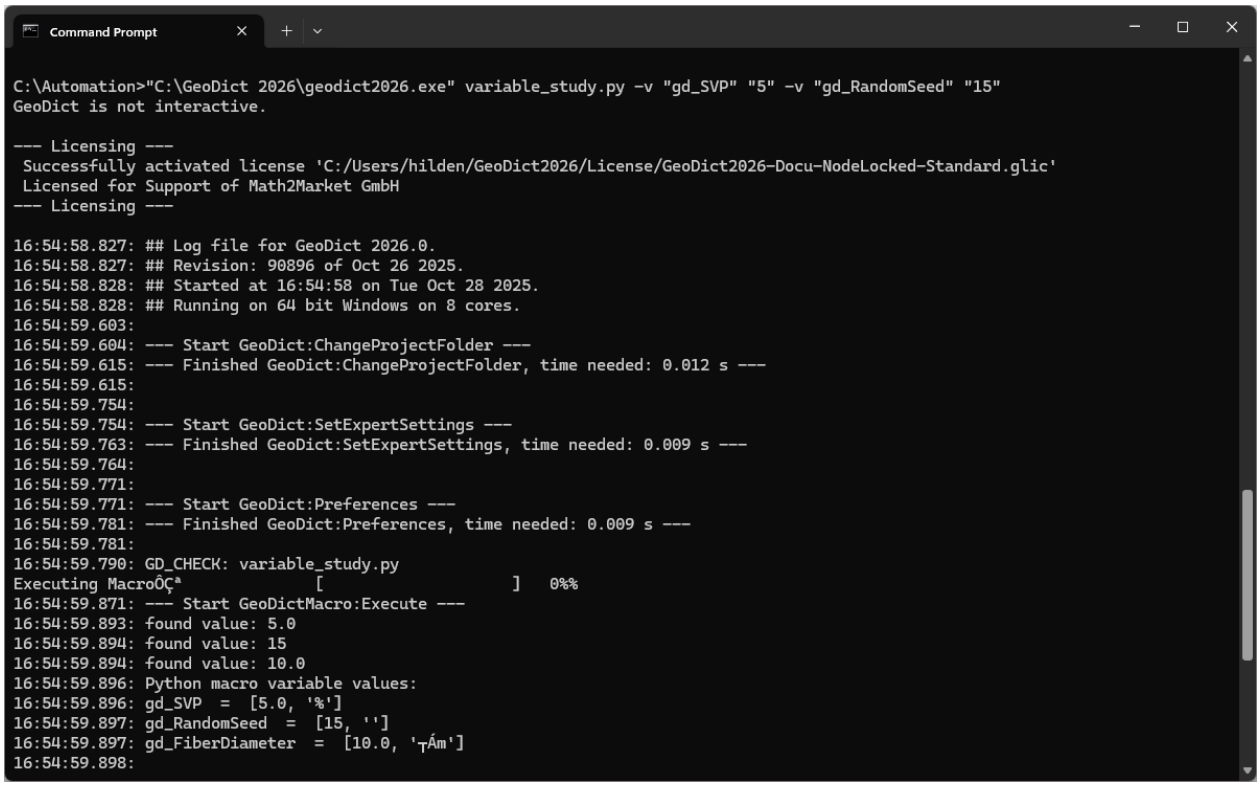
Macros can be executed using the command

>>"Installation-path\GeoDict 2026\geodict2026.exe" macro-file



The result files are stored in the working directory chosen for the command prompt (here C:\Automation), if no other desired file path is given within the macro. If the working directory differs from the macro folder, the file path of the macro also must be given for its execution.

To assign variables from the variables block of a parameter macro use **-v "Key" "Value"** for each variable.



If images should be saved executing a macro, the command **--enable-rendering** is needed. This command opens a hidden GUI until the execution of the macro is terminated. Be aware that you cannot use the **--enable-rendering** option if your command prompt does not support graphical applications, e.g., when you have simply opened a remote SSH connection with PuTTY.

Citation: Math2Market GmbH, 2026: GeoDict 2026 User Guide, GeoPy Scripting,
Math2Market GmbH, Germany, doi.org/10.30423/userguide.geodict

MATH
2 MARKET

Math2Market GmbH
Richard-Wagner-Str. 1, 67655 Kaiserslautern / Germany
www.math2market.com