GeoPy Scripting

to automate GeoDict simulations

User Guide

GeoDict release 2024

Published: February 13, 2024

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

Citation:
Janine Hilden, Jürgen Becker, Anne Blumer, Barbara Planas. GeoDict 2024 User Guide. GeoPy scripting to automate GeoDict simulations handbook. Math2Market GmbH, Germany,
doi.org/10.30423/userguide.geodict

The handbooks in the User Guide series of Math2Market GmbH can be obtained from:

Math2Market GmbH
Richard-Wagner-Strasse 1
67655 Kaiserslautern
Germany

Phone: +49 631 205 605 0
Fax: +49 631 205 605 99
Email: info@math2market.de
Web: www.math2market.de

GeoPy Scripting to Automate GeoDict Simulations

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.

In GeoDict, the older GMC macro language is being phased out and Python is now the language for these scripts.

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 the user's activity that can be reproduced by him/herself 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, the user records several filtration simulations to run them during the weekend or when cluster time is available. Perhaps the user prefers 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. The user 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 the user needs to run GeoDict in a batch queue on a Linux cluster or wants 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, the user 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 the user's own new "effective commands" for GeoDict.

The following lists the most important **definitions** 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. How to record a macro is explained on page 7.
- 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. The structure of a *.py GeoDict macro is described on page 3.
- **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. How to start GeoDict from the command line is described on pages 108ff and how to use GeoDict's own command-line interface is explained on pages 30ff.
- In computer programming **Variables** are used to store information, e.g. in form of numbers (integer, float), text (string) or module parameters (dictionary). The transformation of a simple macro in a parameter macro containing variables is described on pages 36ff.

Further examples and tutorials are found in the **Macro Execution Control**, described on page 8.

There are also helpful workshop videos to be found on the Math2Market YouTube channel.

The **GeoPy for beginners** workshop shows how to record macros, introduce variables and access result files from macros and is split in three parts:

- GeoPy for beginners - Part 1
- GeoPy for beginners – Part 2
- GeoPy for beginners – Part 3

The **GeoPy for advanced users** workshop shows advanced topics as functions, loops, plots, and PowerPoint report generation in three parts:

- GeoPy for advanced users – Part 1
- GeoPy for advanced users – Part 2
- GeoPy for advanced users – Part 3

STRUCTURE OF A GEOPY MACRO (*.PY)

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.
2. **Description = ''' '''** is automatically generated and, before any editing or adding of information, it simply describes the GeoDict version used for recording the macro in the given time and date, and the licensee.
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 about the variables block can be found on page 44.

macro.py

```
Header =  {

Description = '''

Variables = {
   'NumberOfVariables' : 0,
#  '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'
#      },
  }

# 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
# Type:           mandatory, known types are bool, boolgroup, double, uint, int, string, filestring, folderstring, materia
# Unit:           optional,  appears only in GUI (not used to rescale any input parameters automatically)
#                            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 l
#                            for type table,      Unit must be a string with one character per colum, either 'i' (integer)
# 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
```

4. The **command block** contains the commands to be executed by GeoDict.

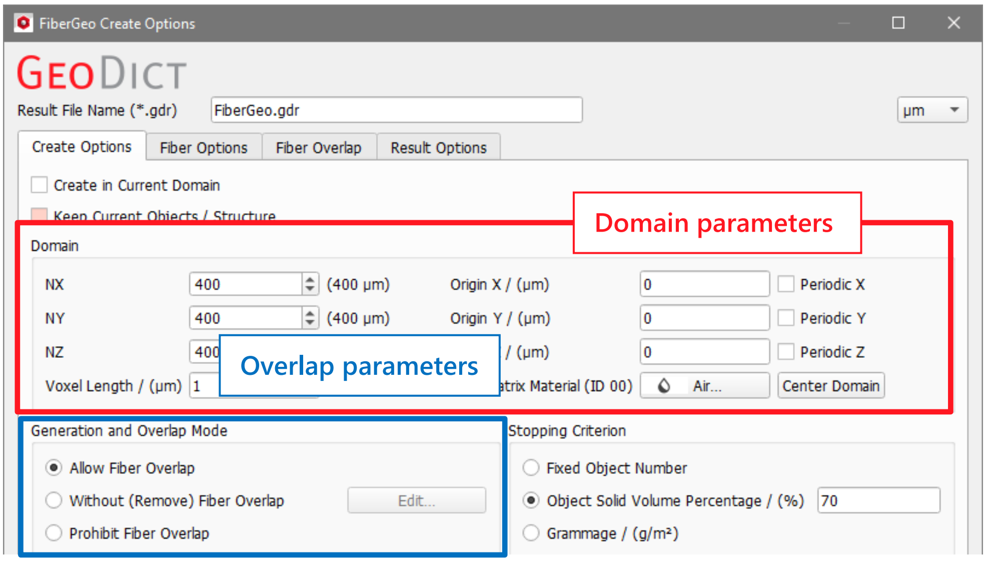
   If **Save macro results to new folder** and **Store general preferences in macro** are checked in the **Start Macro Recording** dialog box (page 7), the block starts with **GeoDict:CreateProjectFolder** and the **GeoDict:Preferences** which are the settings entered in the settings dialog (**Settings → Settings…** in the menu bar).

```
CreateProjectFolder_args_1 = {
gd.runCmd("GeoDict:CreateProjectFolder", CreateProjectFolder_args_1, Header['Release'])

Preferences_args_1 = {
gd.runCmd("GeoDict:Preferences", Preferences_args_1, Header['Release'])
```

   Afterwards 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** 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.



All parameters from the GUI can be found in the recorded macro

```
'MaximalTime'              : (24, 'h'),
'OverlapMode'              : 'AllowOverlap', # Possible
values: AllowOverlap, RemoveOverlap, ForceConnection,
IsolationDistance, ProhibitWithExisting, ProhibitOverlap,
MatchSVFDistribution
'NumberOfObjects'          : 100,
'StoppingCriterion'        : 'SolidVolumePercentage',#
Possible values: SolidVolumePercentage, NumberOfObjects,
Grammage, Density, WeightPercentage, FillToRim, SVP, Number
'SolidVolumePercentage'    : (70, '%'),
'Grammage'                 : (10, 'g/m^2'),
```

```
Create_args_1 = {
  'MaterialMode'             : 'Material',     # Possible
  values: Material, MaterialID
  'MaterialIDMode'           : 'MaterialIDPerObjectType',#
  Possible values: MaterialIDPerObjectType, MaterialIDPerMaterial
  'Domain' : {
    'PeriodicX'           : False,
    'PeriodicY'           : False,
    'PeriodicZ'           : False,
    'OriginX'             : (0, 'm'),
    'OriginY'             : (0, 'm'),
    'OriginZ'             : (0, 'm'),
    'VoxelLength'         : (1e-06, 'm'),
    'DomainMode'          : 'VoxelNumber',
    'NX'                  : 400,
    'NY'                  : 400,
    'NZ'                  : 400,
    'OverlapMode'         : 'GivenMaterial', # Possible values:
    OverlapMaterial, NewMaterial, OldMaterial, GivenMaterial
    'Material' : {
      'Type'        : 'Fluid',
      'Name'        : 'Air',
      'Information' : '',
      },
```

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.

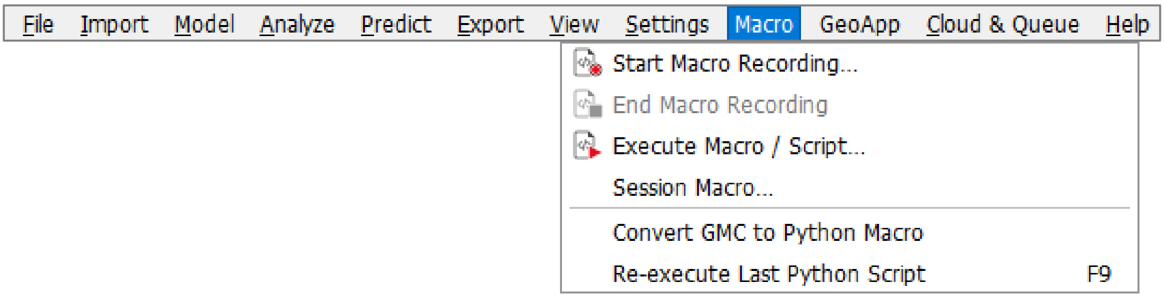
Finally, **gd.runCmd()** executes the command. This function needs three input values:

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

```
  'Generator2' : {
    'Material' : {
      'Type'        : 'Porous',
      'Name'        : 'Manual',
      'Fluid'       : 'Water',
      '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, GivenDirection, InXYPlane,
      AngleAroundDirection, UNDEF
      'DirectionMode' : 'AnisotropyParameter',# Possible values:
      AnisotropyParameter, DirectionTensor
      'Anisotropy1'   : 5,
      'Anisotropy2'   : 1,
      'Phi'           : 0,
      'Theta'         : 0,
      'Psi'           : 0,
      },
    },
  'Temperature'            : (293.15, 'K'),
  }
gd.runCmd("FiberGeo:Create", Create_args_1, Header['Release'])
```

MACRO MENU

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

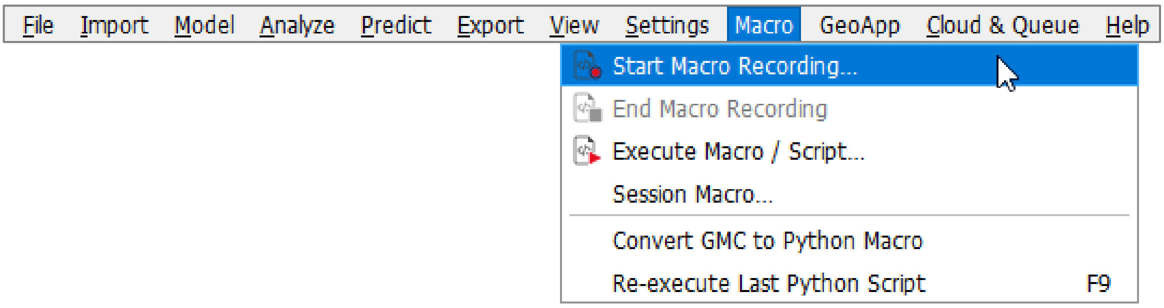


- Recording a macro
- End a macro recording
- Execute a macro or script and access example macros
- Session macro
- Convert GMC macros to Python macros
- Re-execute the last Python script

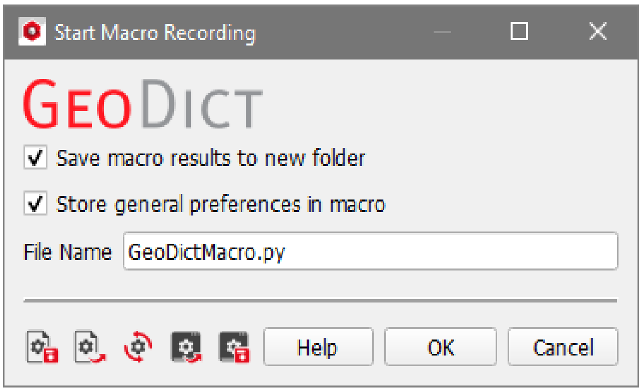
Simple macros are saved while recording a macro or using the **Session Macro** dialog. A **Simple Macro** only contains the recorded commands from the GUI. A simple macro becomes a **Parameter Macro** once variables are defined in it. The macro block listing the variables (**Variables = { }**) is already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

Start Macro Recording

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:



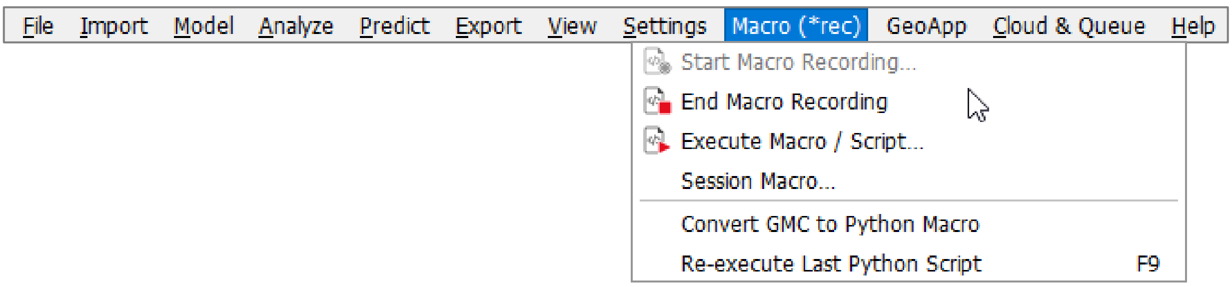
- **Save macro results to new folder** can be selected to include the command **GeoDict:CreateProjectFolder**. The name entered for the macro is given to the newly created project folder. All files created during the execution of the macro are saved in this folder.
- **Store general preferences in macro** can be selected to include the command **GeoDict:Preferences** in the recorded macro (see page 13). 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.

End Macro Recording

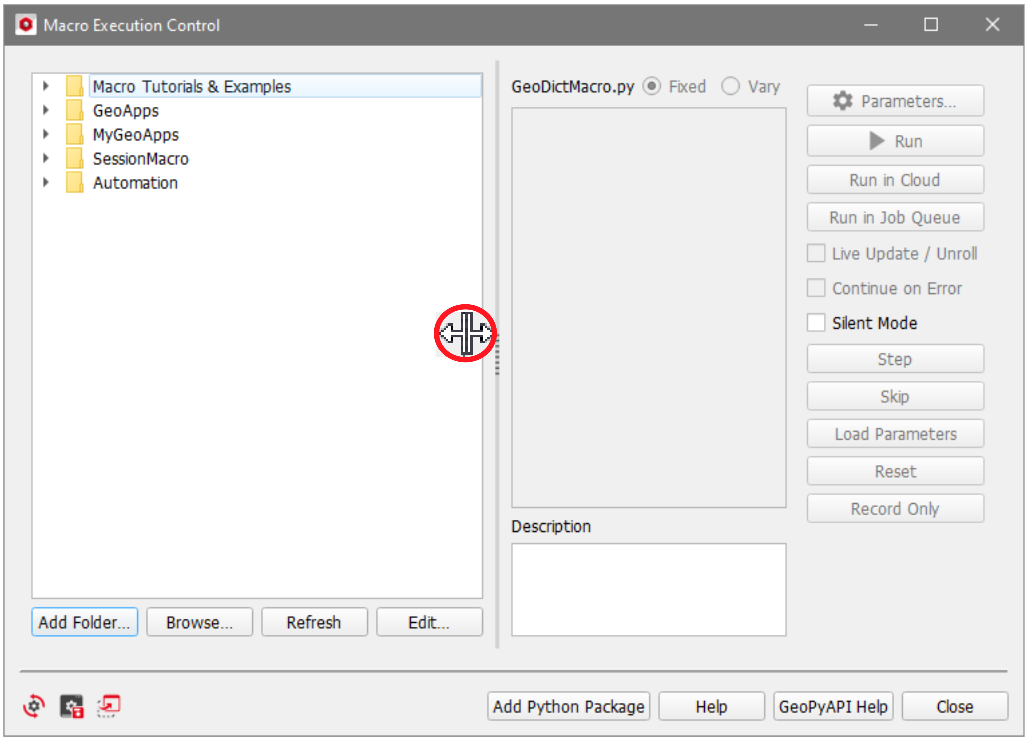
The recording of a macro is stopped by selecting **Macro → End Macro Recording**. This is grayed-out and not selectable unless a macro is being recorded.



Execute Macro / Script

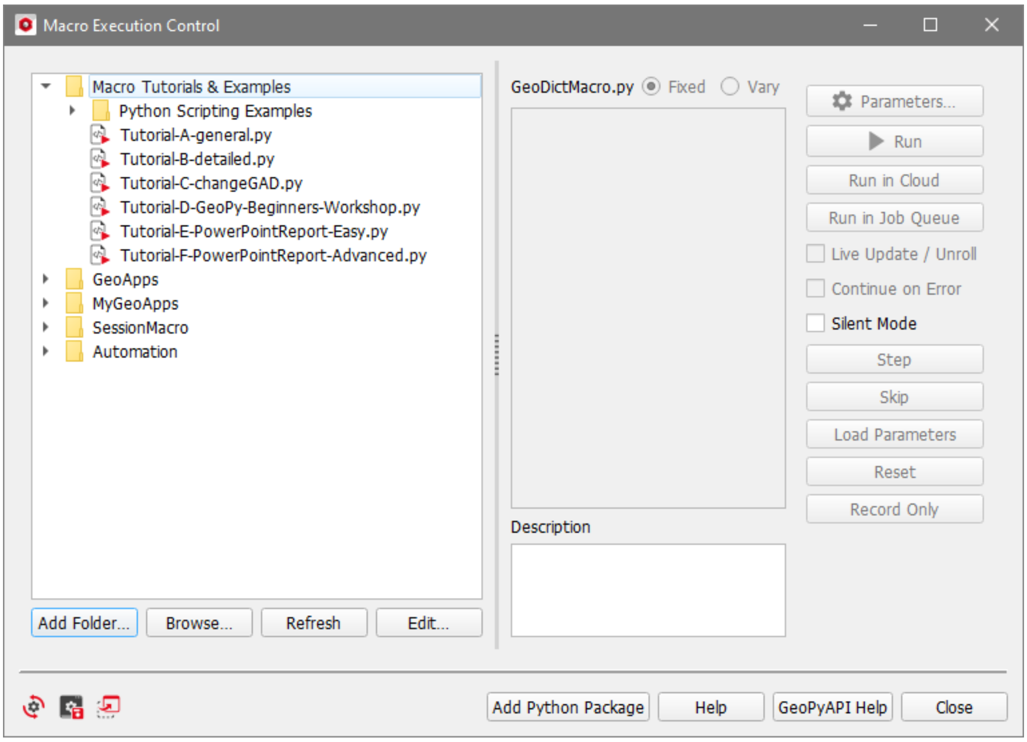
To execute a macro, select **Macro** → **Execute Macro / Script…** to open the **Macro Execution Control** dialog.

The dialog contains two separate parts that can be collapsed and expanded at will.



In the left panel, several folders are listed:

- Preinstalled macros are found by unfolding the **Macro Tutorials & Examples** folder.



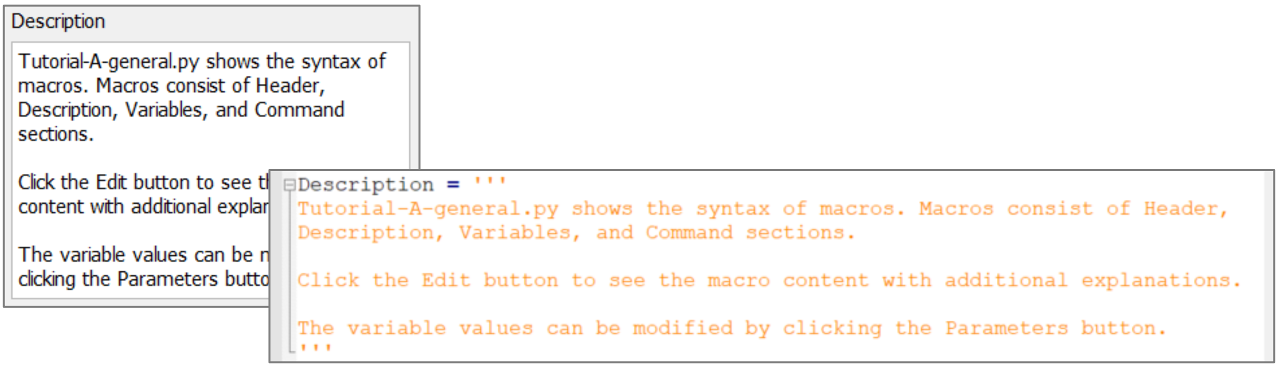
The tutorial macros A, B, C and E need only a GeoDict Base license for execution. The tutorials D and F also need the modules FiberGeo and FlowDict and are the macros created in the workshop videos available on the Math2Market YouTube channel. Find the corresponding links in the macros by opening them in a text editor clicking **Edit** as described on page 10 or use the links given on page 2.

All tutorials have detailed descriptions and thus can be very helpful for getting started with editing Python macros.

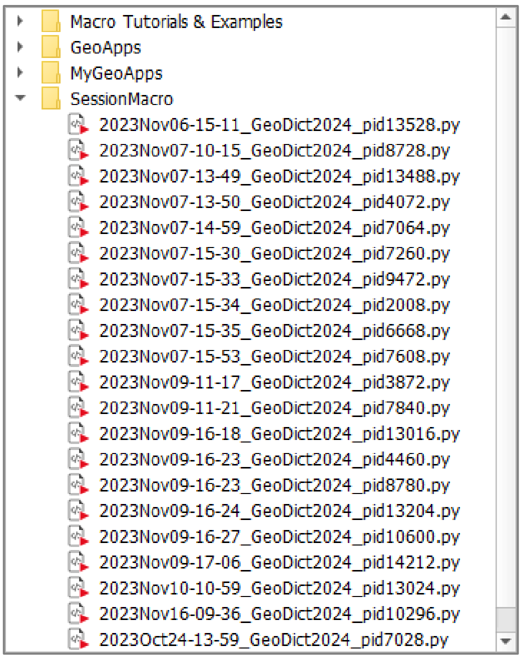
More advanced example macros can be found in the subfolder **Python Scripting Examples.** These Python scripts also use other GeoDict modules.

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.

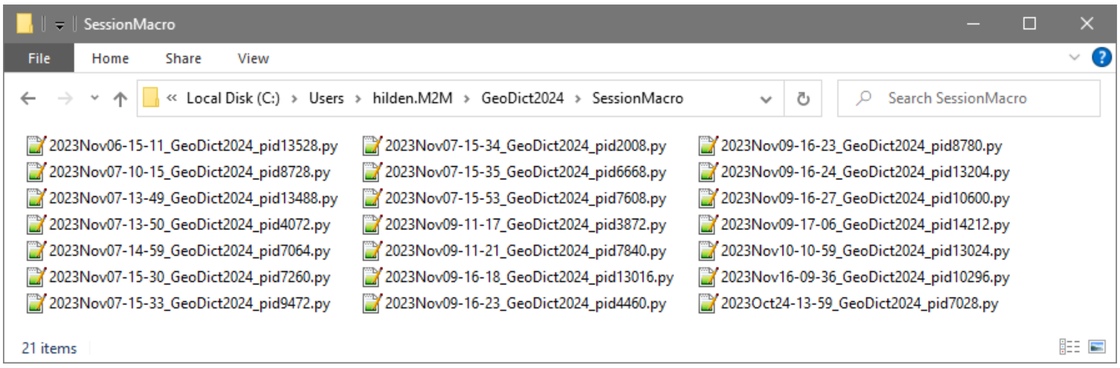
For the **Tutorial-A-general.py** macro, the text in the macro and in the description area are shown here.



- The **GeoApps** folder contains all the GeoApps to be found by selecting **GeoApp** from the menu bar. They are described in the GeoApps handbook of this User Guide.
- The **MyGeoApps** folder can be filled with the user's own GeoApps as also described in the GeoApps handbook. By default, it contains MyFirstGeoApp, an example GeoApp
- 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** dialog described on pages 24ff.



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



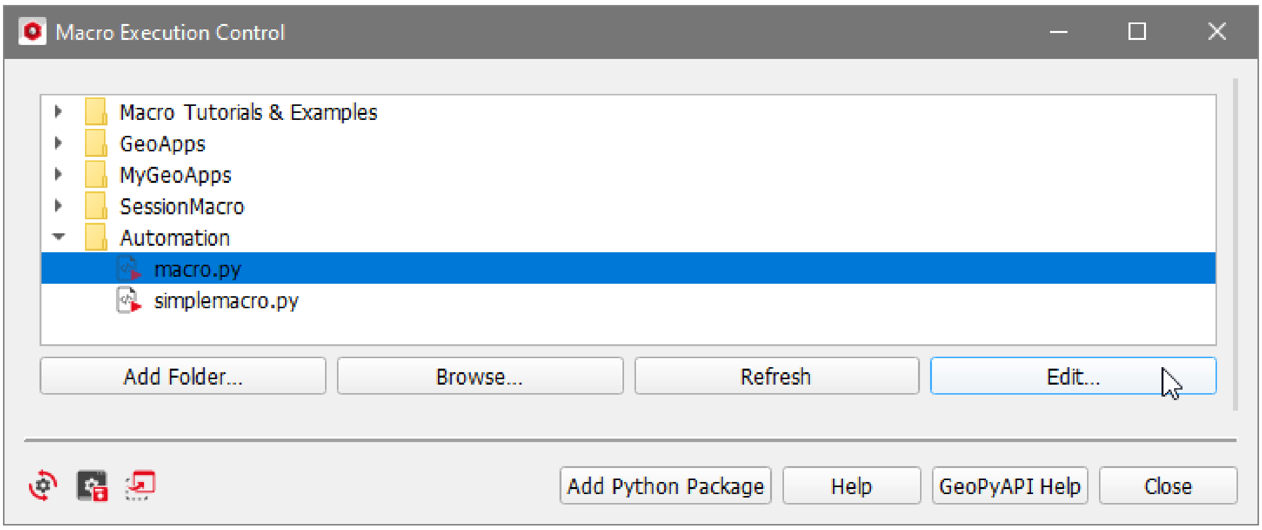
- The last folder is the selected project folder. All macros inside this folder are shown. Fill this folder for example, by using **Record Macro** (described on page 7) or the **Session Macro** dialog (described on pages 24ff).

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 for example in the folders **GeoApps**, **GrainGeo**, **Macro Tutorials & Examples** or **GeoDictAI** included in the installation folder of GeoDict.
- **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++.

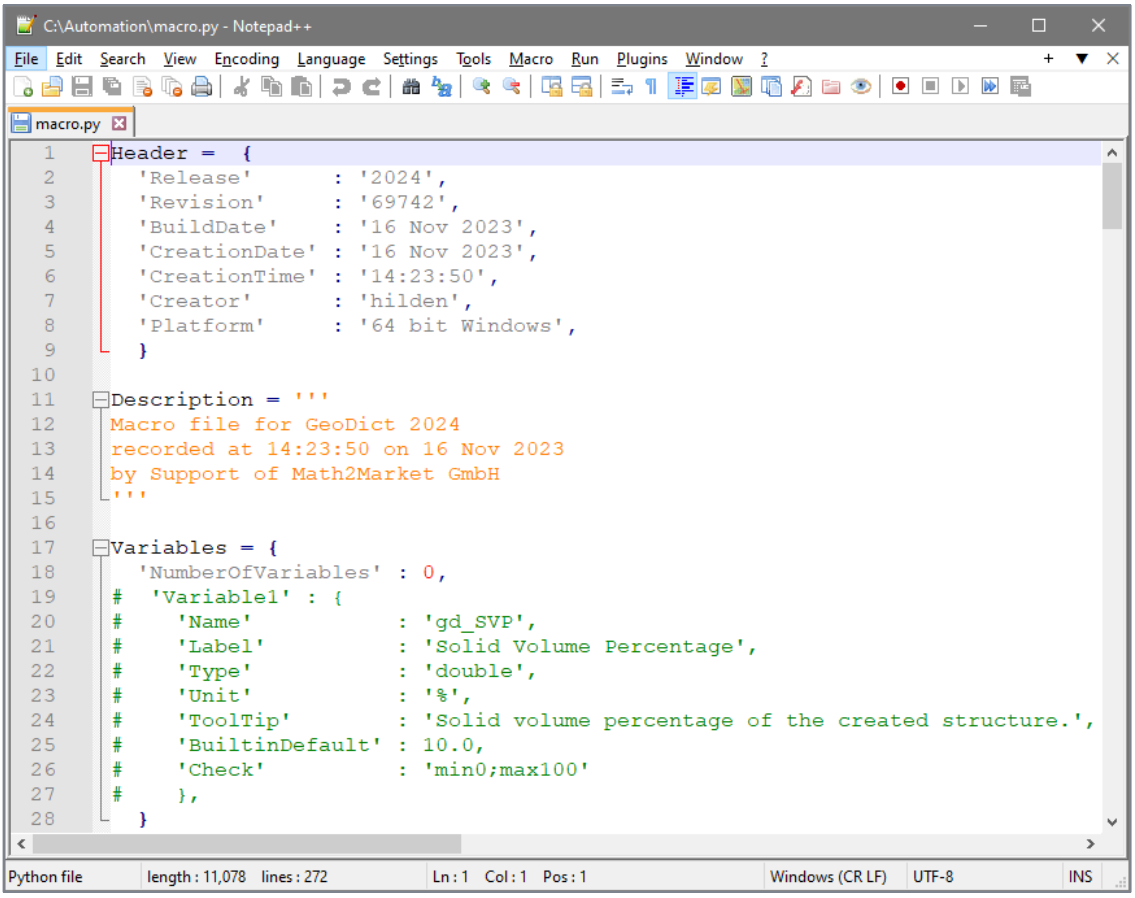
  The basic way to edit a macro (e.g. **macro.py**), is to find the macro file name in the project folder, right click on it, and select **Open With...**. Choose the editor from the list of available programs. However, the **macro.py** can be directly opened, and then edited from the **Macro Execution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

  Click **Edit...** to open the selected macro using the designated text editor (see page 34 on how to set it). The macro then can be examined and edited.

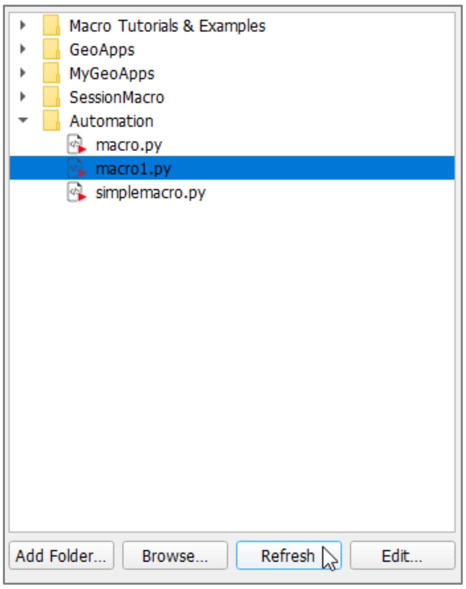


The macro follows the structure explained on page 3: Header={}, Description=''' ''', Variables={} and the command block.

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



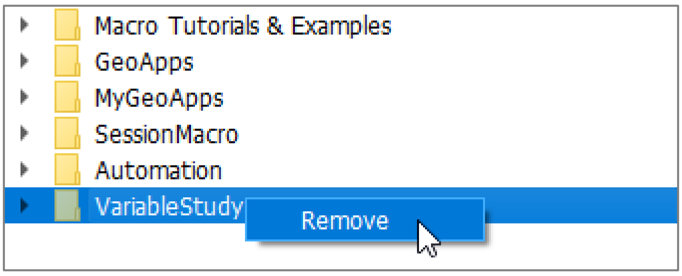
After modifications, the macro file can be saved with a different name (e.g. macro1.py). Click **Refresh** to have the name of the macro, modified and saved in the project folder, appear in the list of macros in the left panel of the **Macro Execution Control** dialog.



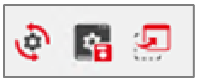
GeoDict does not recognize a file as a macro when the file extension is not *.py or *.gmc. This can happen for example when Windows settings are such that extensions are not shown and, coincidentally the text editor (i.e. Editor or WordPad) automatically adds an extension to the file name (*.txt, *.doc, 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. How to set a text editor as default editor is described on page 34.

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**.

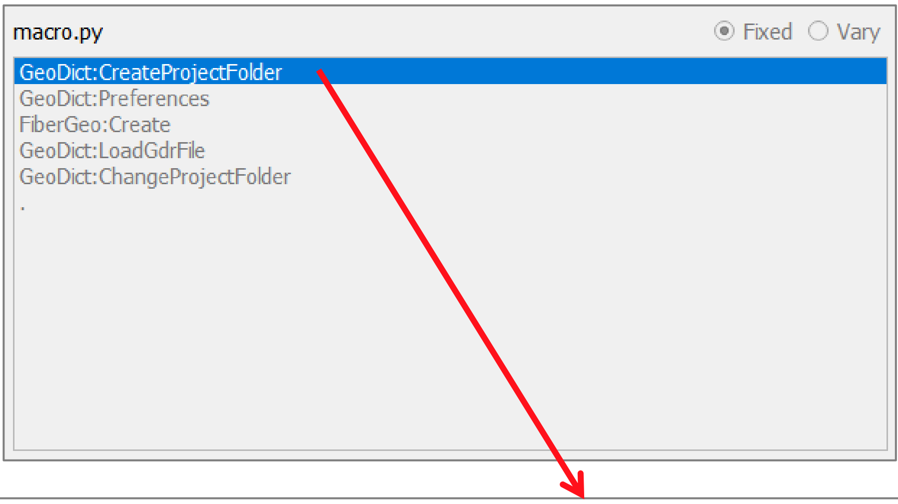


Load the built-in default folders, set the current folders as start-up settings or raise the GeoDict main window through the icons at the bottom left of the dialog when needed. Resting the mouse pointer over an icon prompts a tooltip showing the icon's function to appear.



Macro Description

On the right part of the **Macro Execution Control** dialog, the entries in the upper panel correspond to each one of the gd.runCmd() (see page 50) commands, that can be seen when opening the macro with a text editor.




```
CreateProjectFolder_args_1 = {
    'FolderName' : 'macro',
  }
gd.runCmd("GeoDict:CreateProjectFolder", CreateProjectFolder_args_1, Header['Release'])
```

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

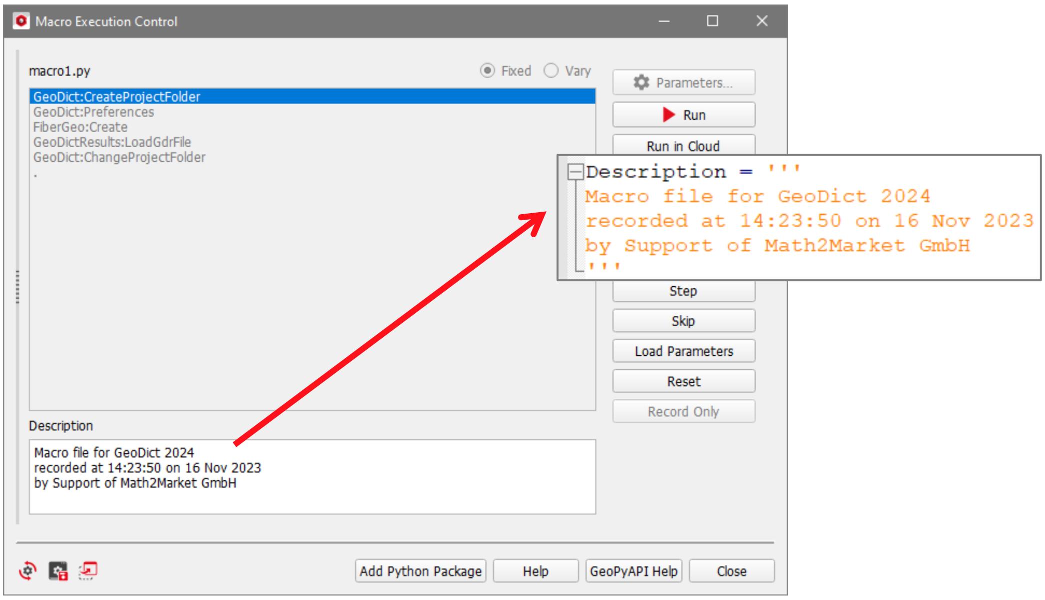
```
    'IO' : {
      'WriteDbgPrintfToConsole' : False,
      'WriteDbgPrintfToLogfile' : True,
      'UseWaitingTime'          : True,
      'WaitingTime'             : 20,
      'LogFileStorageTime'      : 21,
      },
    'TextEditorFullPath'        : 'C:/Program Files/Notepad++/notepad++.exe',
    'Undo' : {
      'Memory' : (2048, 'MiB'),
      },
    'Update' : {
      'CheckForUpdates' : True,
      },
    }
gd.runCmd("GeoDict:Preferences", Preferences_args_1, Header['Release'])
```

```
      'OrientationDistribution' : {
        'Type'            : 'AnisotropicDirection',# Possible values: Isotropic, Ani
        'DirectionMode'   : 'AnisotropyParameter',# Possible values: AnisotropyParam
        'Anisotropy1'     : 5,
        'Anisotropy2'     : 1,
        'Phi'             : 0,
        'Theta'           : 0,
        'Psi'             : 0,
        },
      },
    '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'])
```

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.

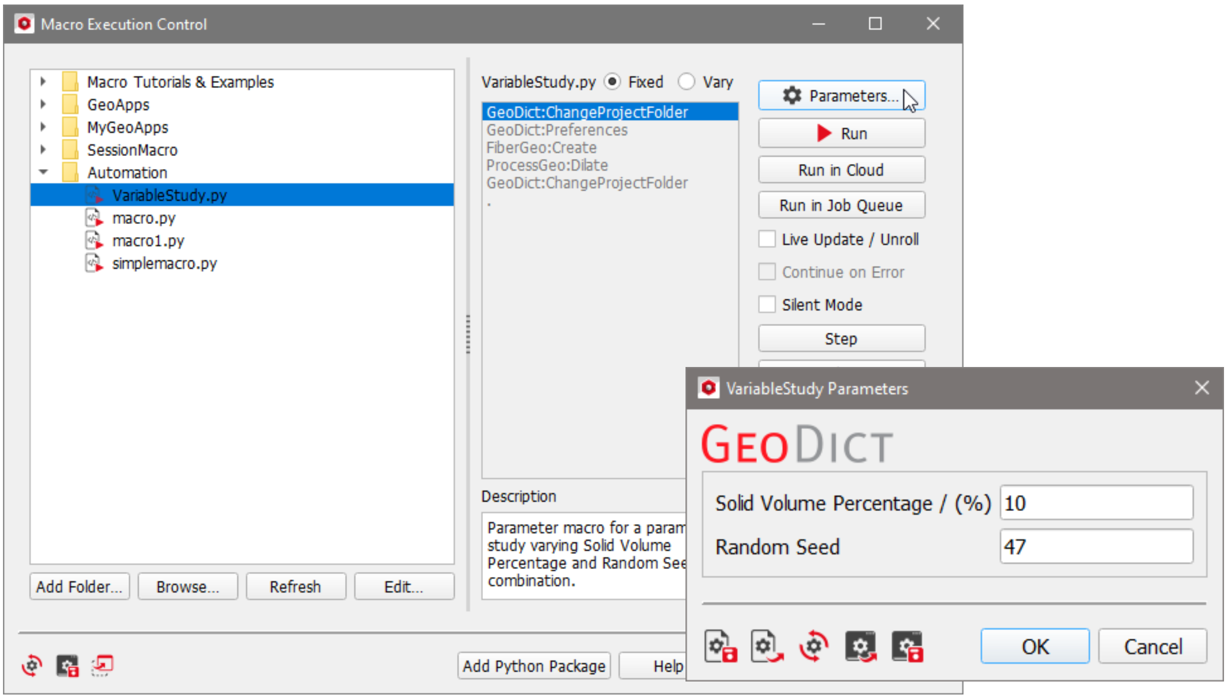


Fixed and Vary Parameters

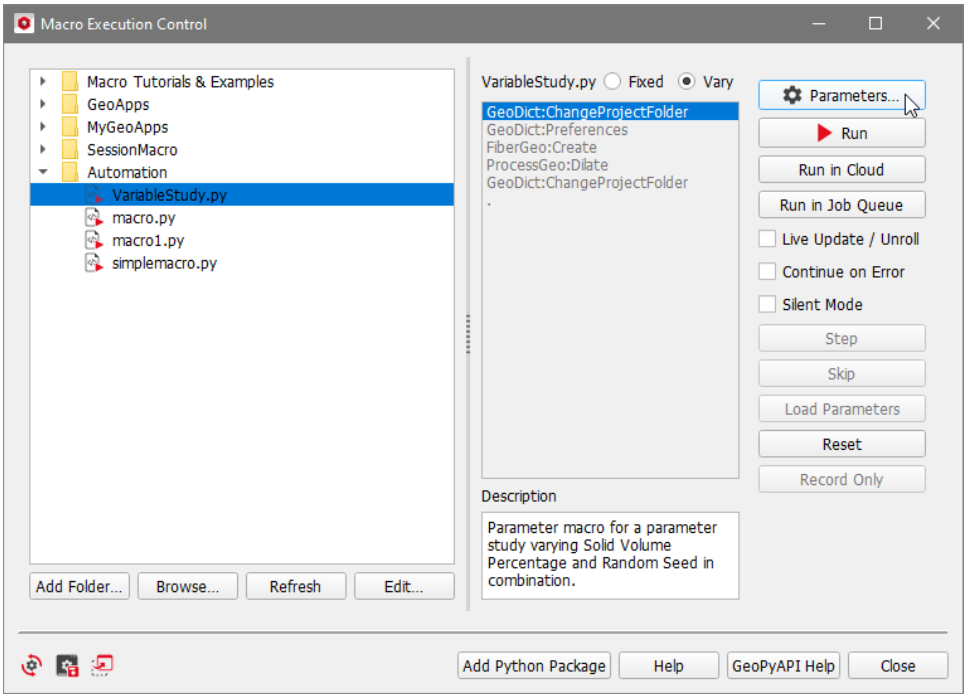
For the user's convenience, the macro block listing the variables (Variables = {}) is already created during the recording of a simple macro, but it is initially empty of variables. A simple macro can be transformed into a parameter macro as explained below starting on page 36.

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.

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

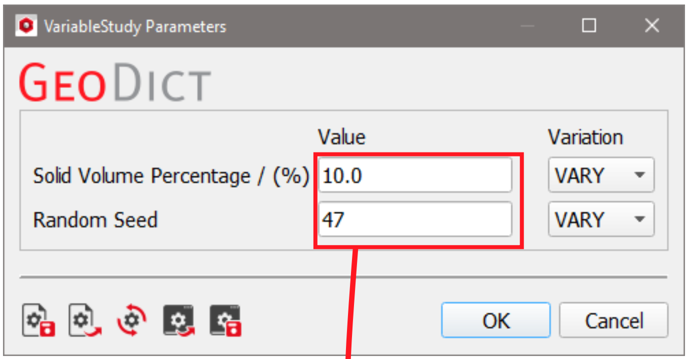


With **Vary** selected, clicking **Parameters** opens a different parameter dialog box where parameter lists can be entered.



The macro is executed several times with different parameter values combinations.

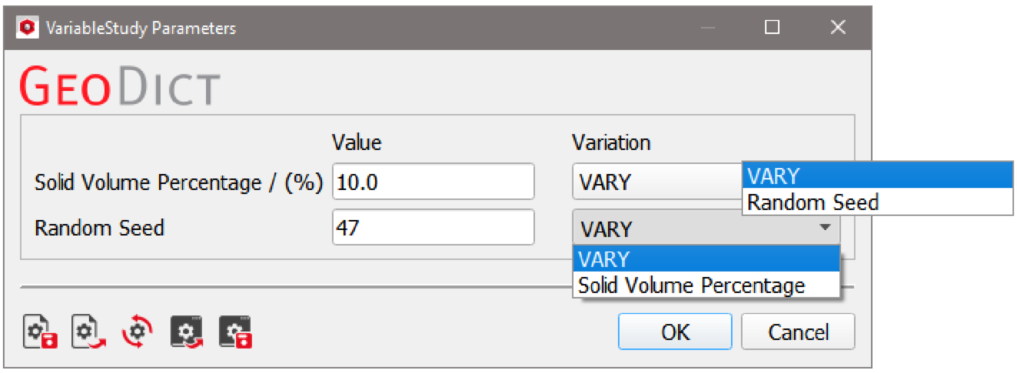
In the macro, the variable values are described within the brackets of **Variables = {}**. They are listed right after the header.




```
Variables = {
  'NumberOfVariables' : 2,
  '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',
    'ToolTip'        : 'Random Seed of the created structure.',
    'BuiltinDefault' : 47,
    },
  }
```

In the **VariableStudy.py** macro, two variables are present as indicated by the line **'NumberOfVariables' : 2**. 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 **47** here). Learn more about the different variable types on page 44.

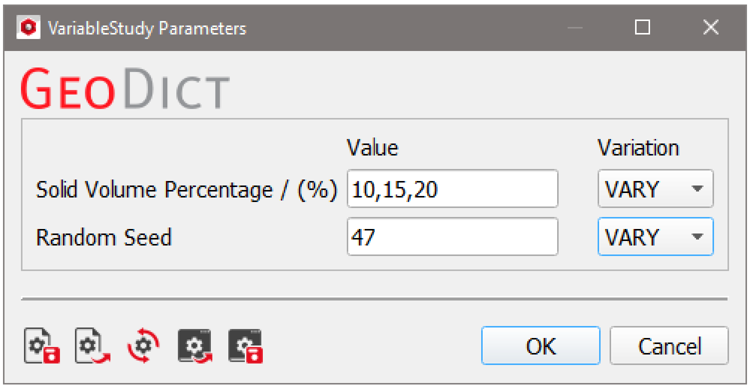
When editing a parameter macro to run a parameter study in which several variable values should be tried out, the **Value** and the **Variation** for each of the variables must be set. The **Variation** can be set to **VARY** for a list of variable values or can be coupled to another variable. Coupled variables are run in a synchronized way. When the value of one variable is varied, the value of the coupled variable is modified accordingly.



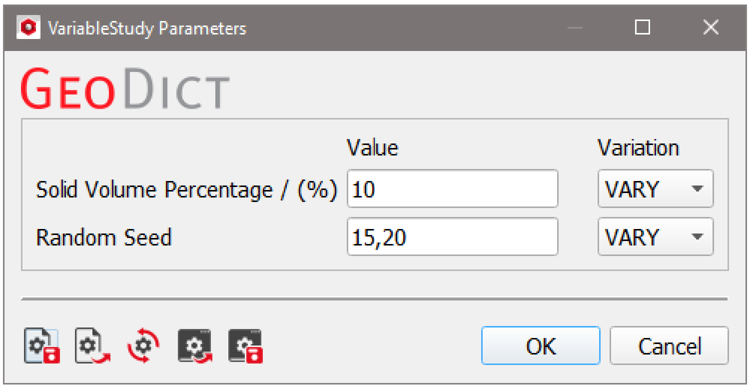
To couple variables, the same number of values must be entered under **Value** in the boxes for every variable.

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

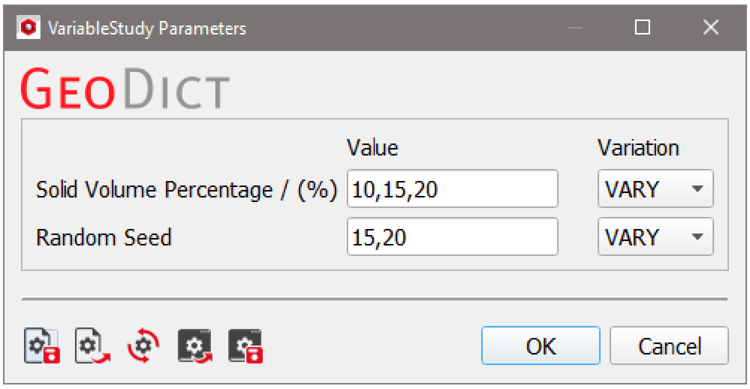
All possible combinations of the **Solid Volume Percentage** values with the single **Random Seed** value are executed, leading to runs with variable values **(10,47)**, **(15,47)**, **(20,47)**. The value of the second variable is kept constant



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)**.

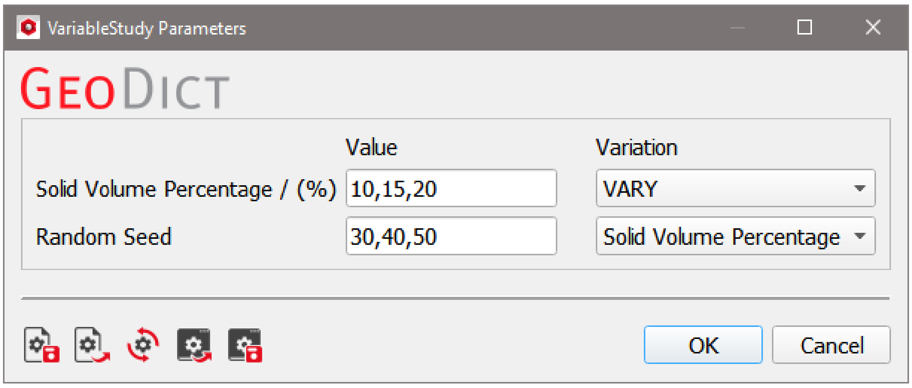


In the following case all possible combinations of the two **Random Seed** values with the three **Solid Volume Percentage** value are executed, leading to pairs **(10,15)** and **(10,20)**, **(15,15)**, **(15,20)**, **(20,15)**, **(20,20)**.

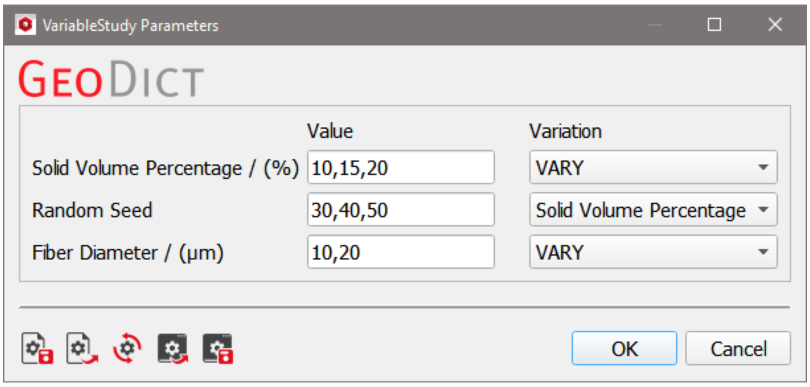


Setting the parameter **Variation** to the other parameter leads to coupled pairs. As mentioned in page 16, the same number of values for every variable must be entered in the boxes.

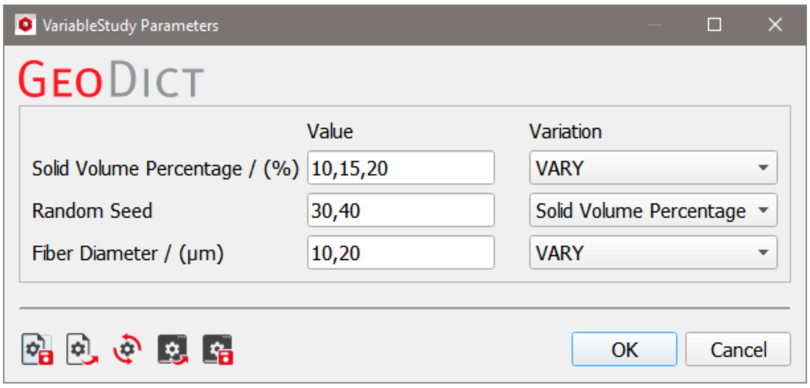
The first values in **Solid Volume Percentage** (**10**) and **Random Seed** (**30**) are coupled with each other, as well as the second values with each other (**15** and **40**), and the third values with each other (**20** and **50**), resulting in the combinations **(10,30)**, **(15,40)**, and **(20,50)**.

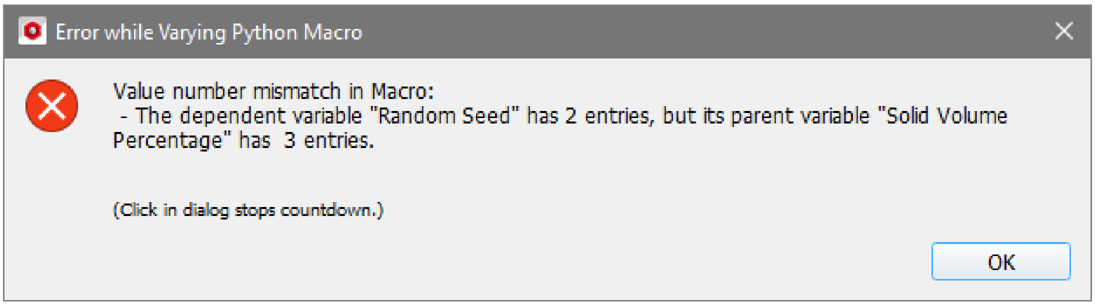


If a parameter macro contains more than two variables, not all variables must be coupled. Coupling **Random Seed** to **Solid Volume Percentage** and leaving **Fiber Diameter** to **VARY**, leads to the combinations **(10,30,10)**, **(10,30,20)**, **(15,40,10)**, **(15,40,20)**, **(20,50,10)** and **(20,50,20)**.



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

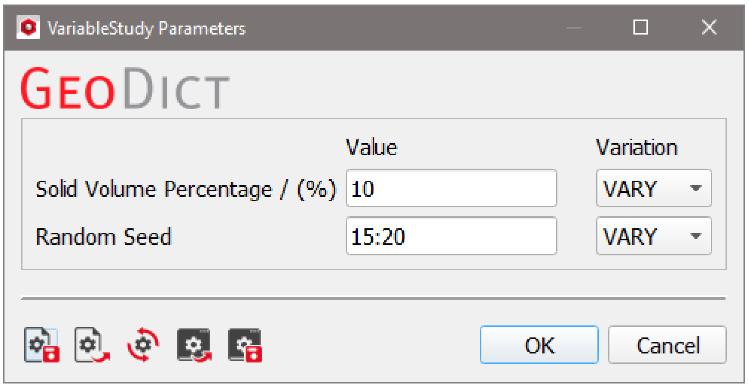




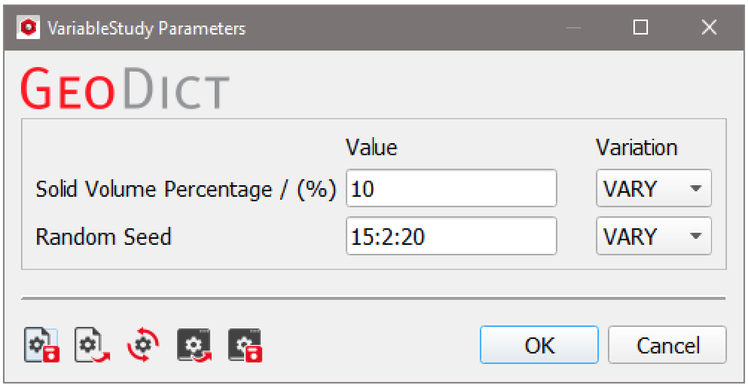
Otherwise, clicking **Run** starts the execution of the parameter macro.

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.

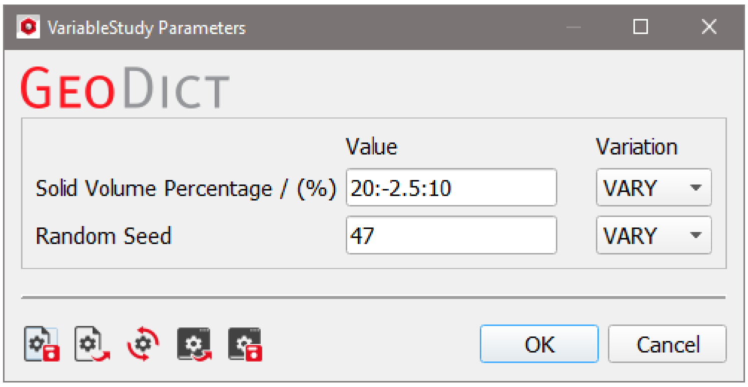
The notation **15:20** means that all the values between 15 and 20 are taken for the computation. This results in the combinations **(10,15)**, **(10,16)**, **(10,17)**, **(10,18)**, **(10,19)**, and **(10,20)**.



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 2.5th 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)**.

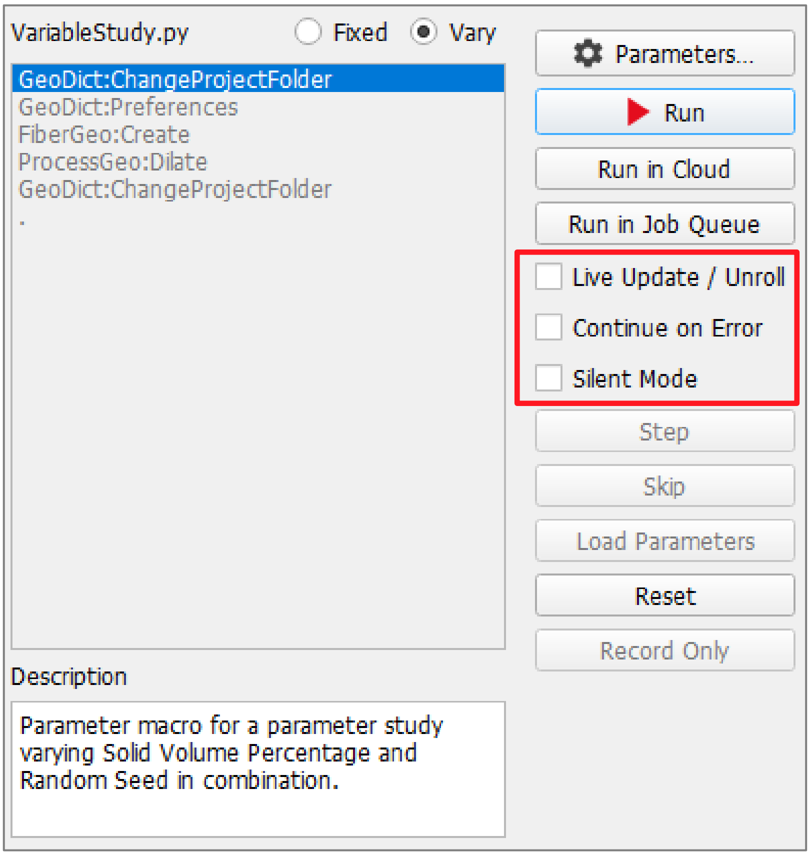


Run (in Cloud, in Job Queue), Live Update, Continue on Error, Silent Mode, Step, Skip, Load Parameters, Reset macro and Record Only

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 High Performance Computing handbook of the User Guide 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 High Performance Computing handbook of the User Guide. If interested in cloud simulations or job queueing contact Math2Market to apply for a GeoDict cloud or job queueing 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, instead of only recording the GeoDictMacro:Execute command. However, the execution of the macro is faster if this box stays unchecked.

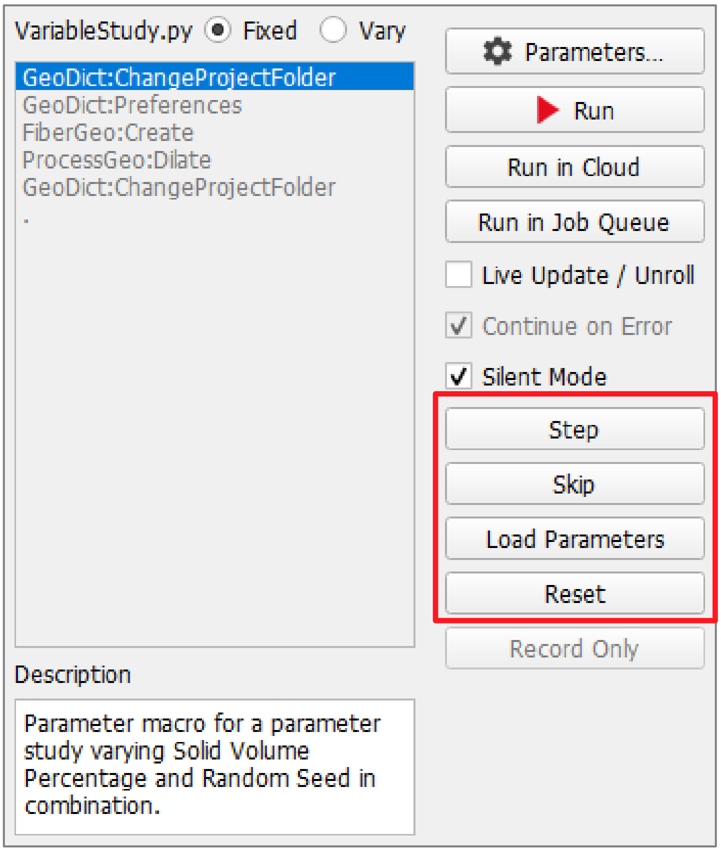


The **Continue on Error** checkbox below can only be checked if **Vary** is selected. Check **Continue on Error** to execute all parameter combinations entered to the **Parameter** dialog box that work and not only all up to the parameter that results in an error.

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.

If **Silent Mode** is checked, no message boxes are shown during the macro execution.

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

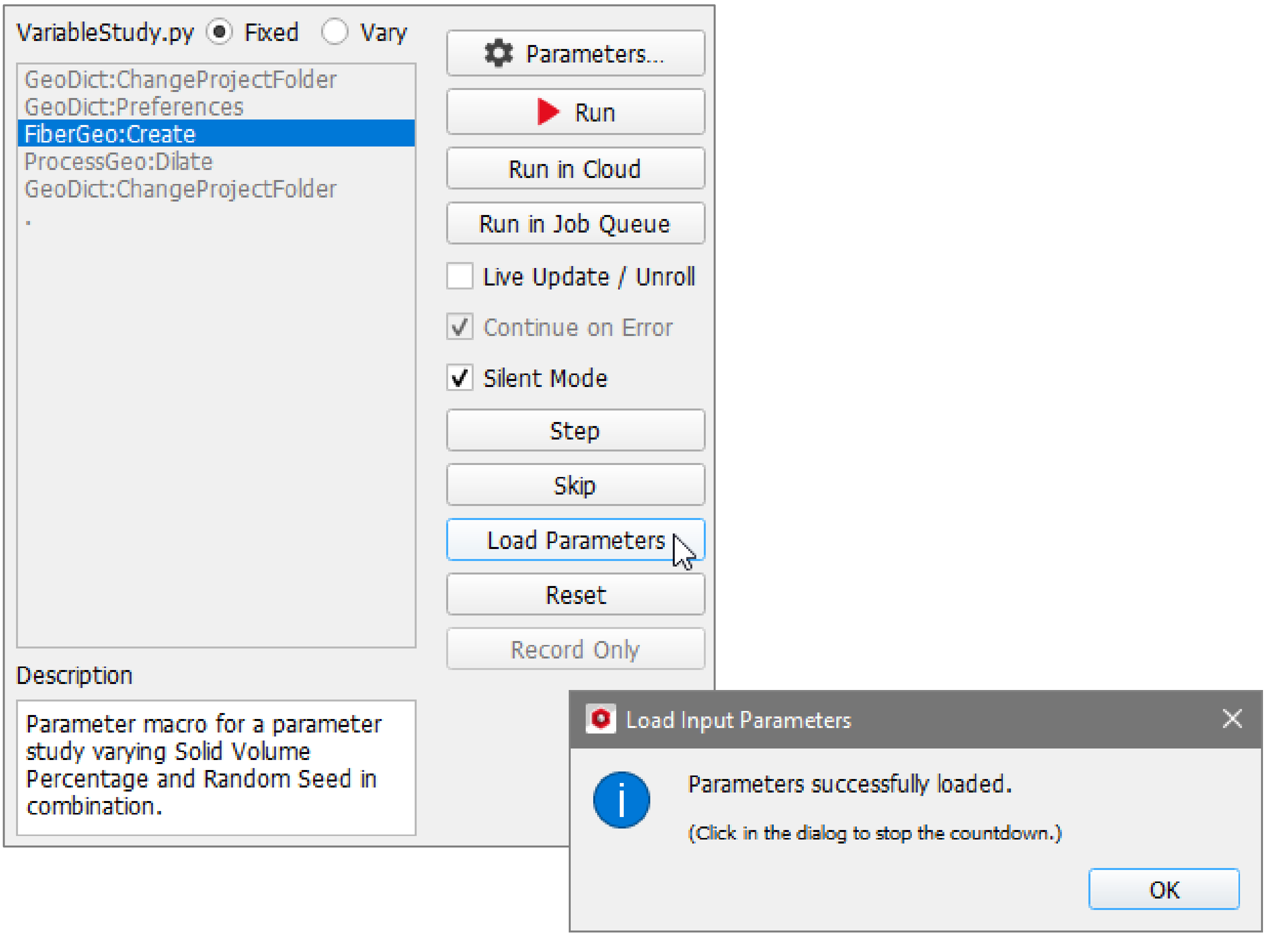
The user must consider the consequences that the skipping of a command has. For example, an error message appears when skipping the creation of a new project folder for the data, so that the data is actually saved in the current project folder and then, trying to leave the (not created and not existing)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

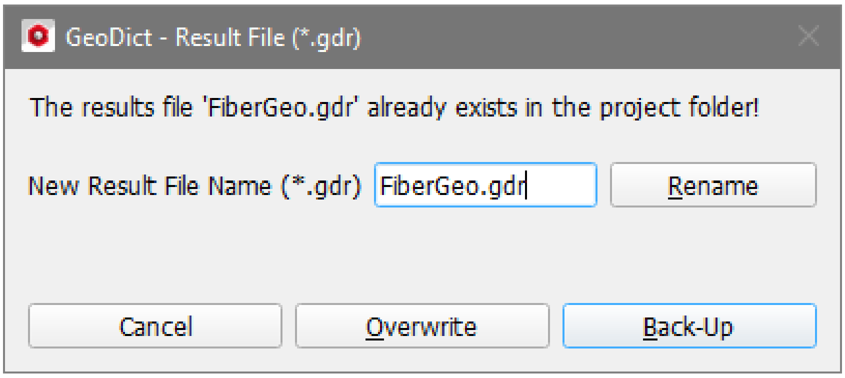
Loading the parameters might be interesting if the user decides to abandon the execution of the macro at a given command, and to post-process the structure by modifying its parameters directly in the module's GUI, to obtain a different result.



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.

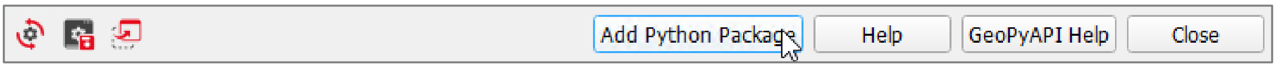
Click **Record Only** while recording a macro to record the commands and the variables edited in the **Parametes** dialog of the selected macro in the **Macro Execution Control**.

When the executed macro includes a command for which the user 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** these message do not appear and the results are back-upped automatically.

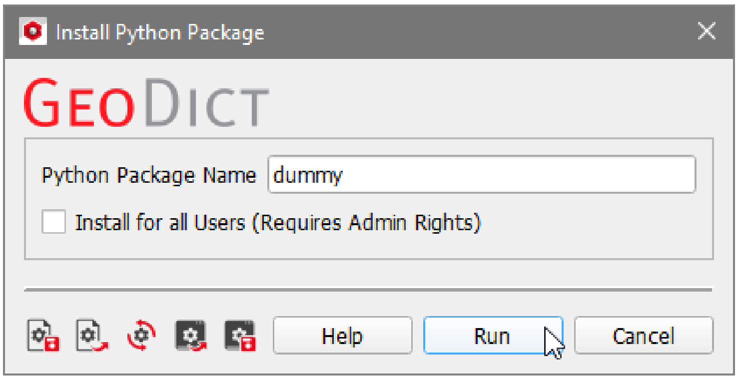


Adding other Python packages

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 2024\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\GeoDict2024\Python). After installation, restart GeoDict to use the new package.



It is also possible, to install needed Python packages offline, if downloaded before. For this, run a Python macro as described on page 19. 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 on page 7. 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'    : 'dummy',                      # 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)
```

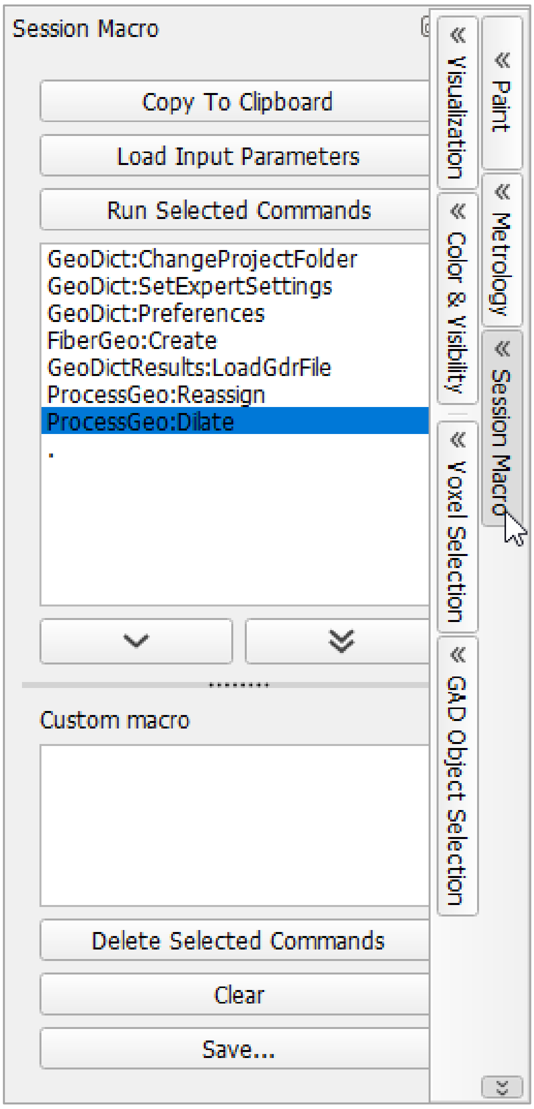
GeoPyAPI Help

Click **GeoPyAPI Help** to open an overview about all GeoDict Python API commands, described on pages 50ff. This overview is opened as an *.html file in the default browser.

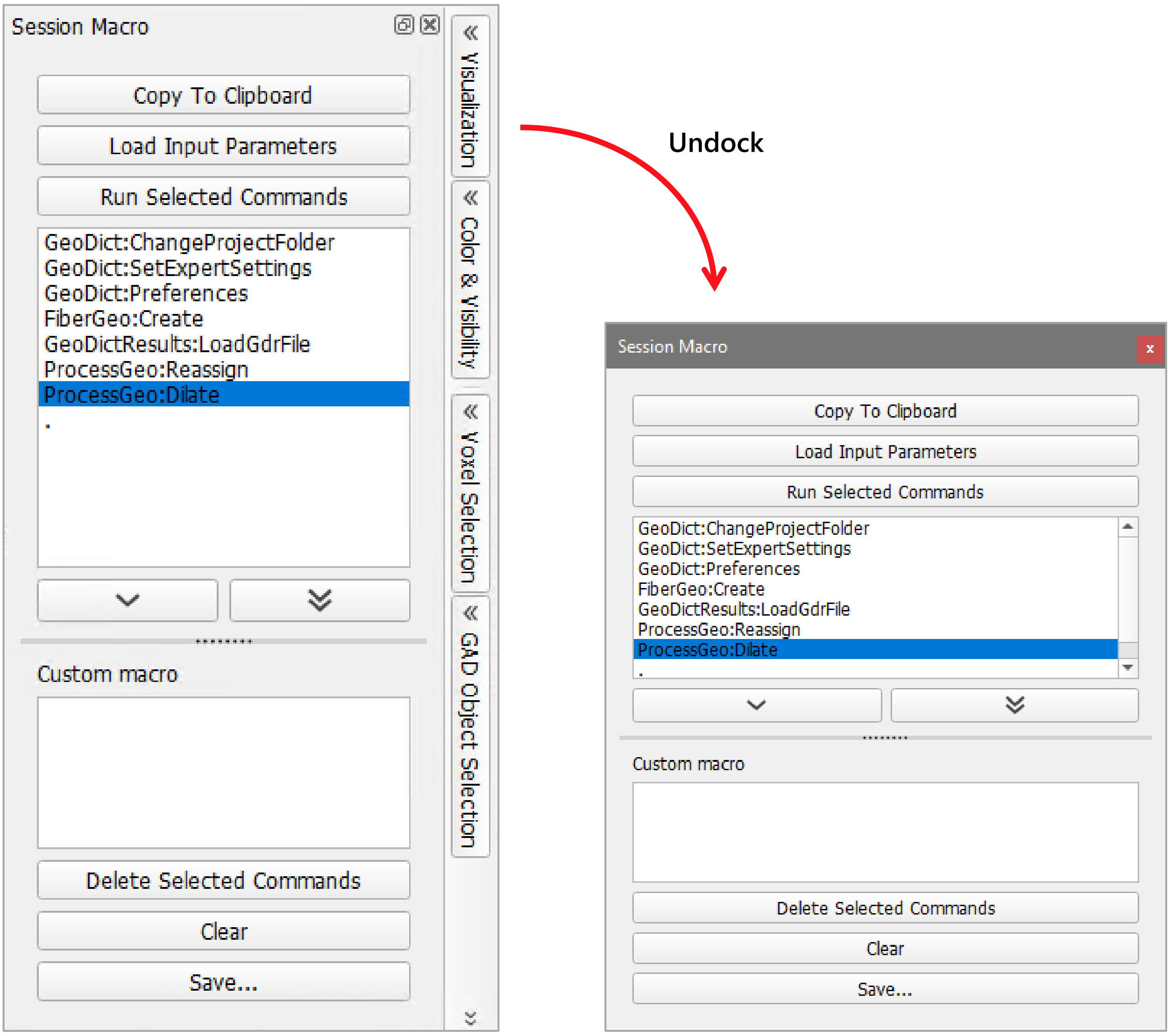
Session macro

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

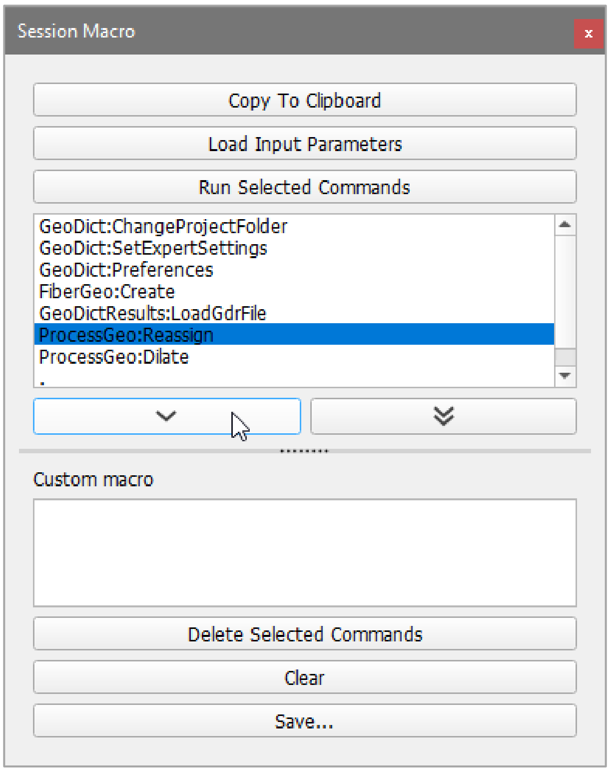
After selecting **Macro** → **Session Macro…** in the menu bar, the **Session Macro** side bar opens. It can also be opened by clicking on the side bar tab on the right of GeoDict. If the GeoDict window is not maximized, the Session Macro tab may not be shown. It can be found by clicking ⌄ at the bottom of the side bar.

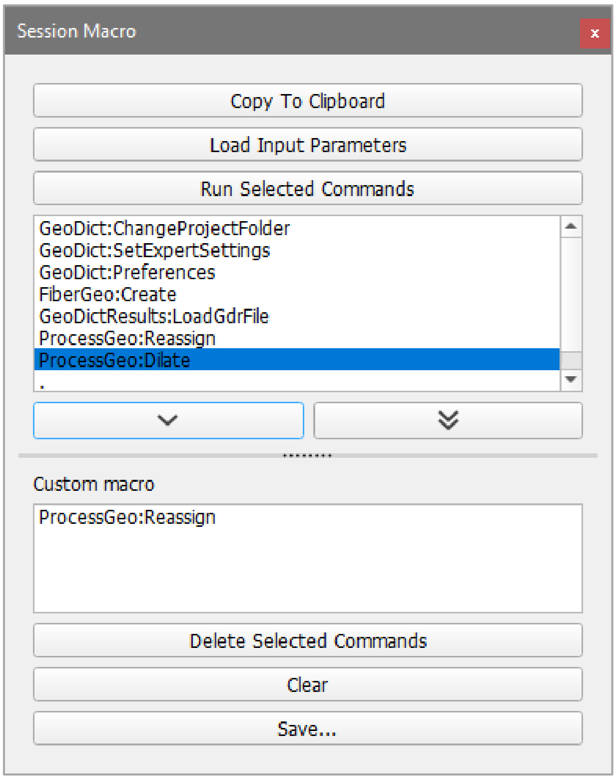


Undock the **Session Macro** panel and turn it into the **Session Macro** dialog by clicking ⧉ in the upper right corner. Although it is still minimized if the GeoDict GUI is minimized, the dialog can be moved independently on the screen.



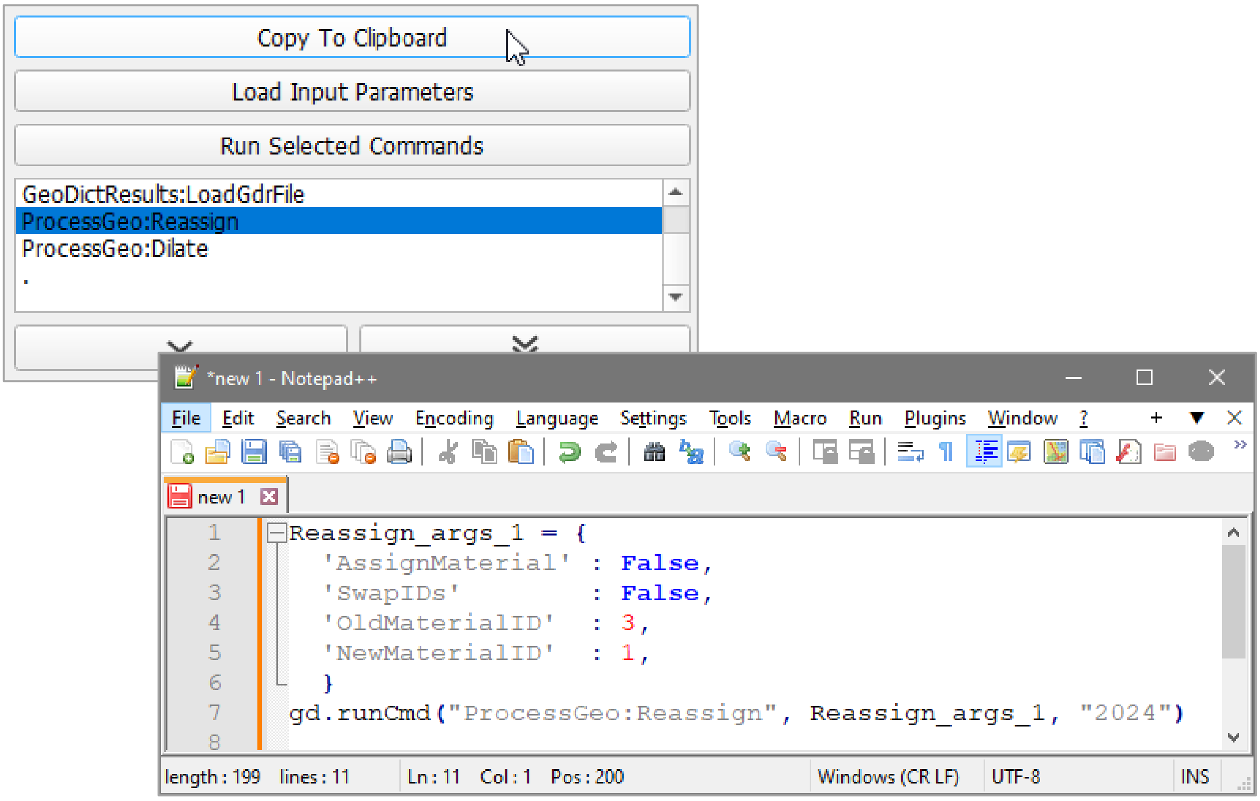
The commands used during the session appear in the upper panel of the **Session Macro** dialog and can be selected (highlighted). Click the single arrow  to move the selected commands to the lower panel in the desired order.



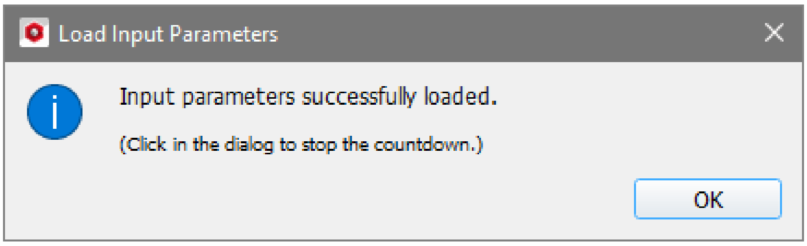


To choose and move all commands from the upper panel at once, click the double arrows  instead.

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

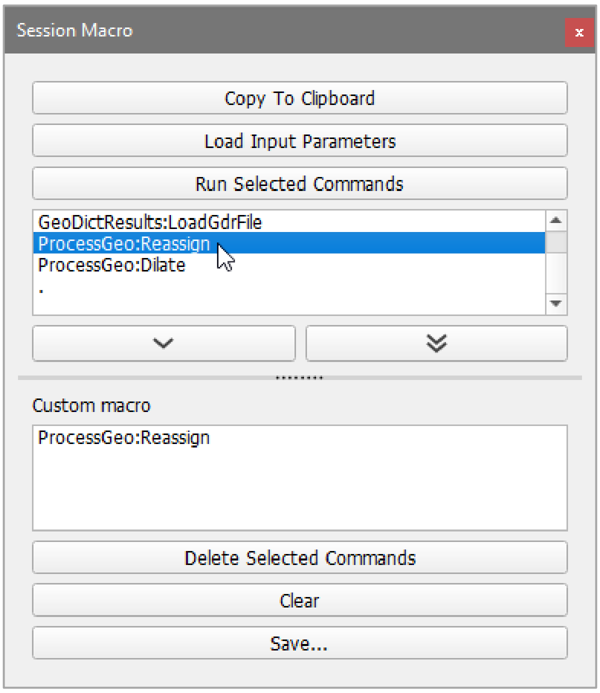


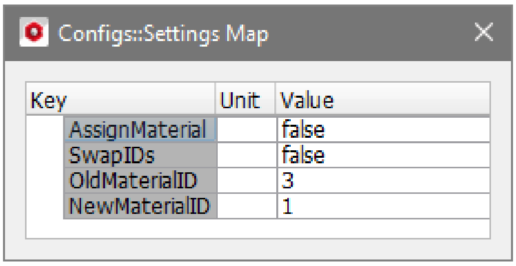
Click **Load Input Parameters** to only load the parameters of a single highlighted command in the corresponding parameters dialog box in the module section.



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

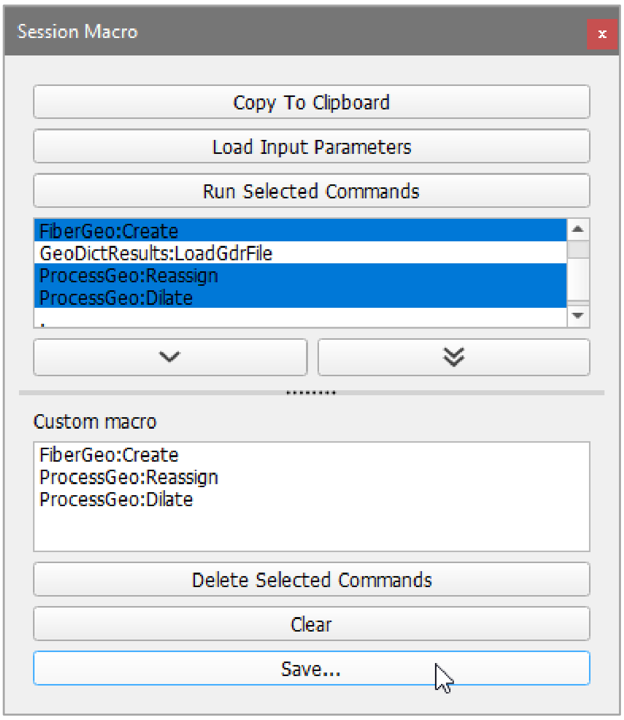
Double clicking on a command, whether in the upper or in the lower panel, shows the corresponding settings map in a new dialog.



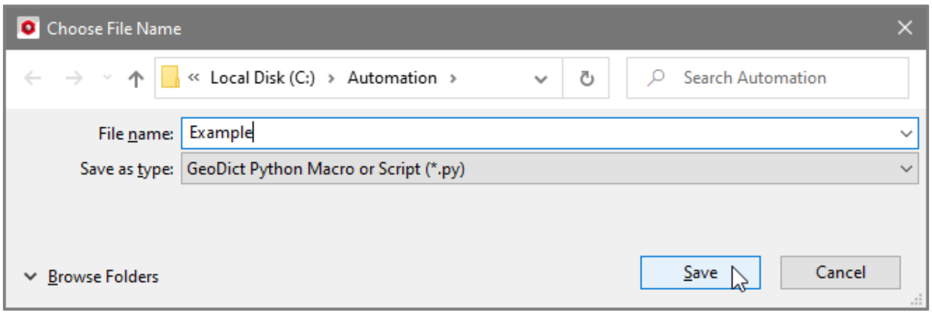


The commands can be removed from the lower panel by highlighting them and clicking **Delete Selected Commands**. To remove all commands at once, click **Clear.**

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



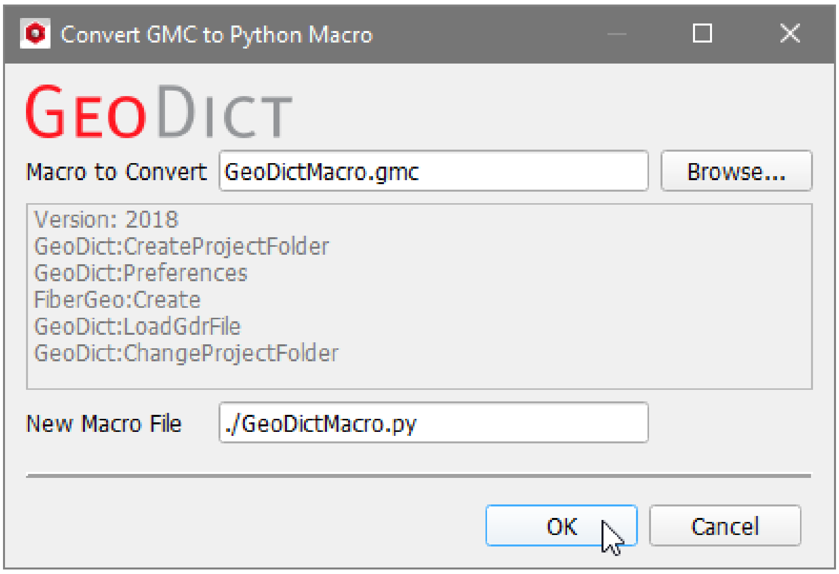
In the appearing dialog, choose a filename and the desired folder where the macro will be stored.



Convert GMC to Python Macro

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



Click **OK**. The new Python macro can be found in the same folder as the GMC macro.

Re-execute the last Python script.

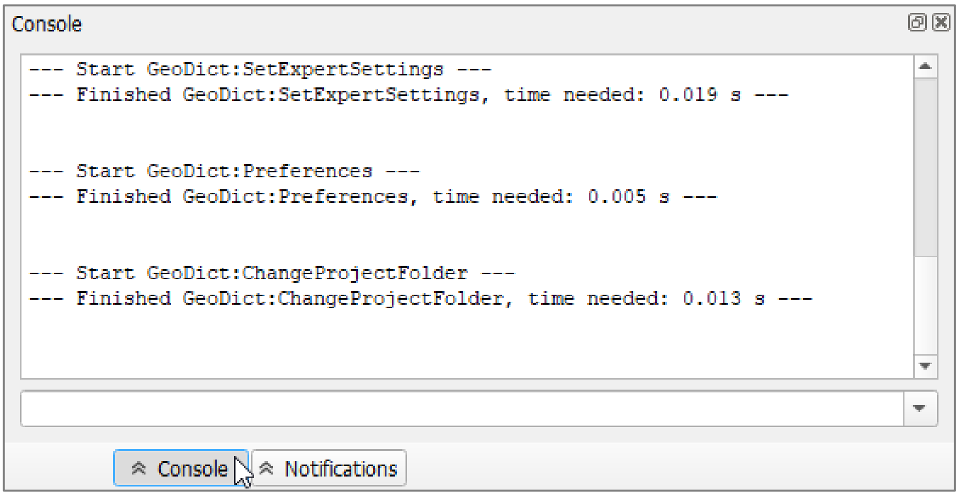
To quickly execute again the last Python script, select **Macro → Re-execute Last Python Script** or press F9.

The python script is simply executed again without other selections.

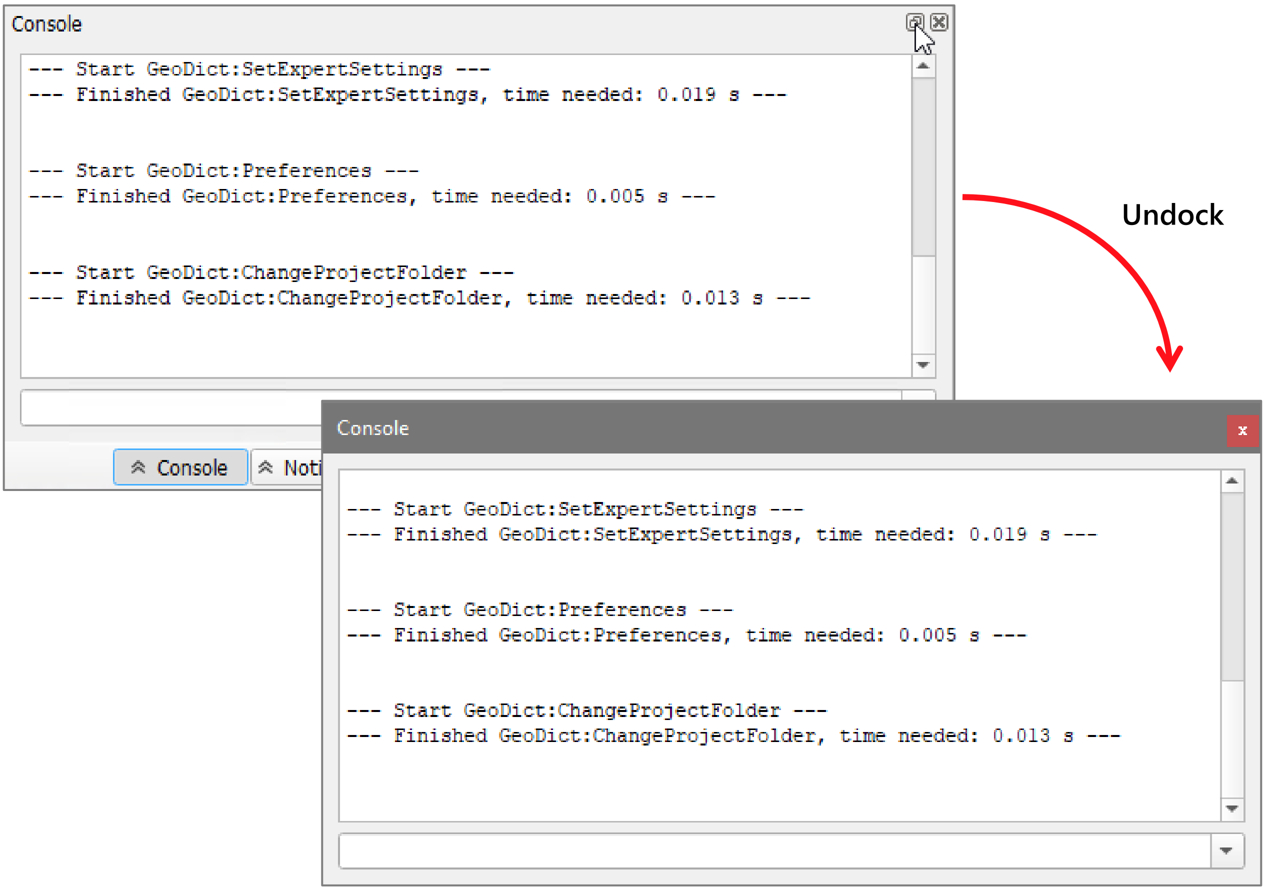
GeoDict Console

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

The console is found in the GeoDict GUI below the visualization area. This section can be folded and unfolded by clicking on **Console** (Console) in the bottom of GeoDict.

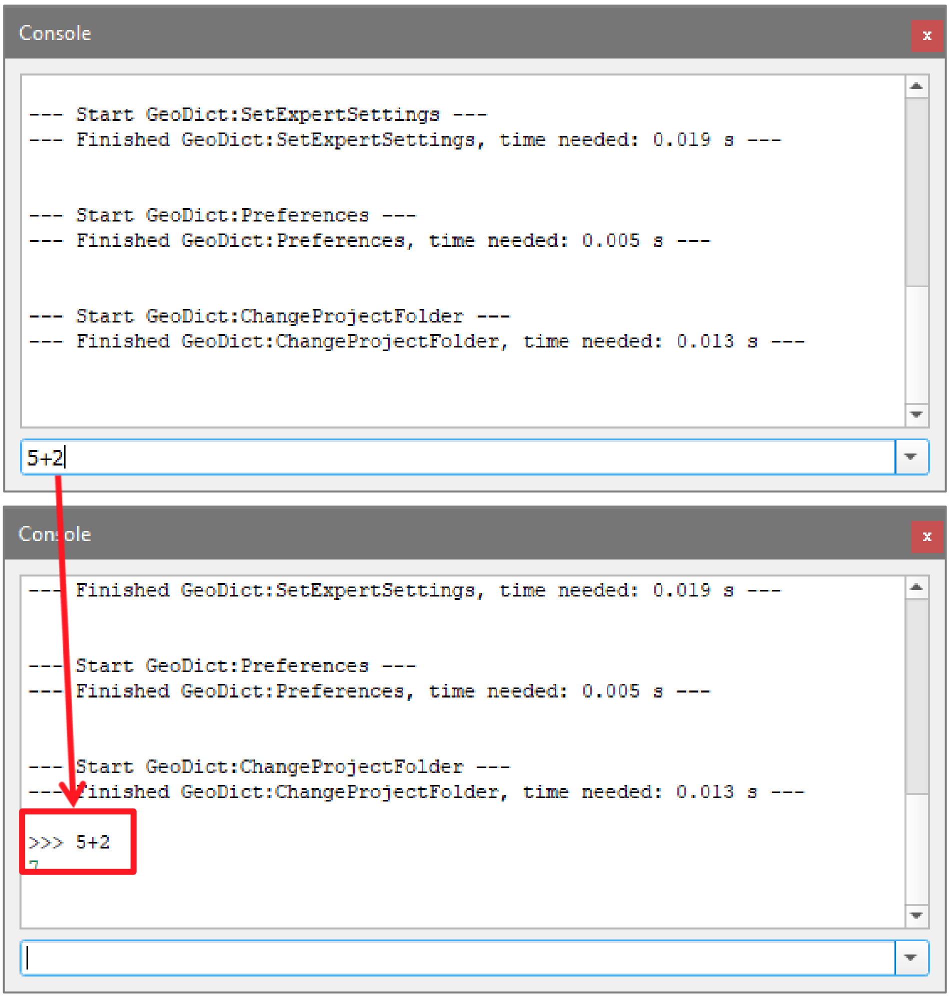


Clicking the symbol in the upper right corner, separates or undocks the console from the rest of the GUI. Although it is still minimized if the GeoDict GUI is minimized, the dialog can be moved independently on the screen.

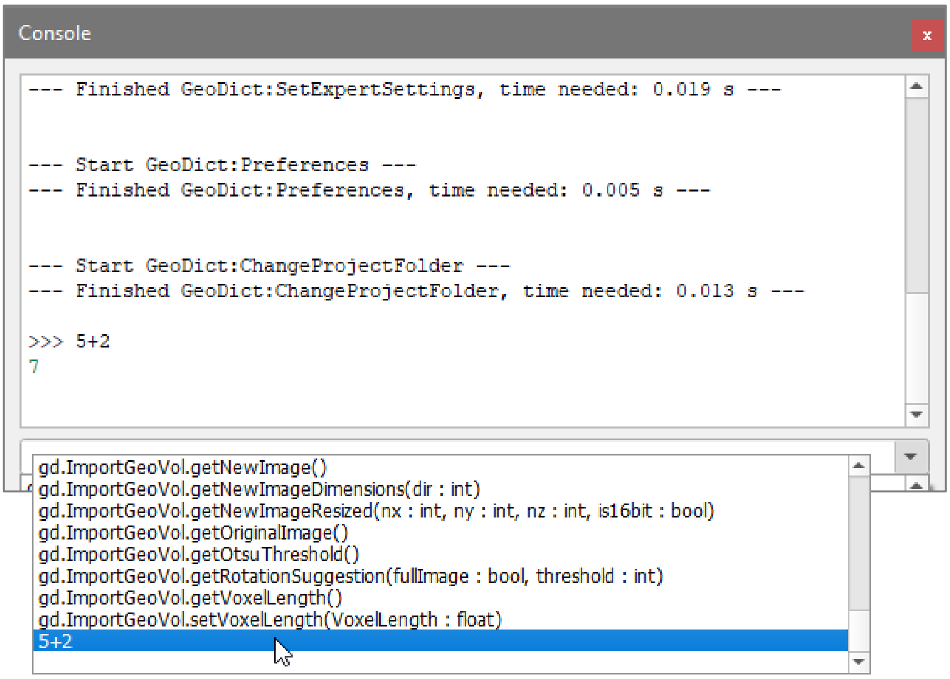


To connect the console with the GUI again, drag and drop it to its place at the bottom of the GUI or simply close the dialog.

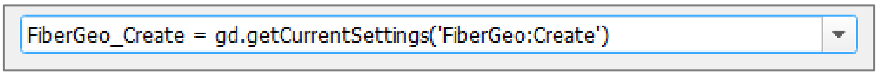
The box below the console can be used to run Python commands. One command line at a time can be inserted, and it is run by pressing **Enter** on the keyboard.



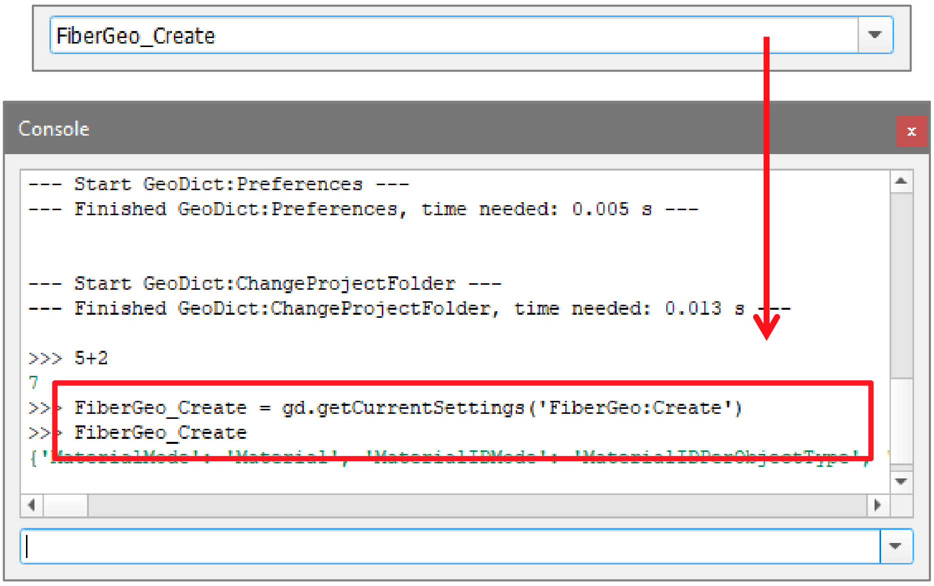
Unfolding the pull-down menu of the box shows the last used commands and some standard commands from the GeoDict Python API described on pages 50f.



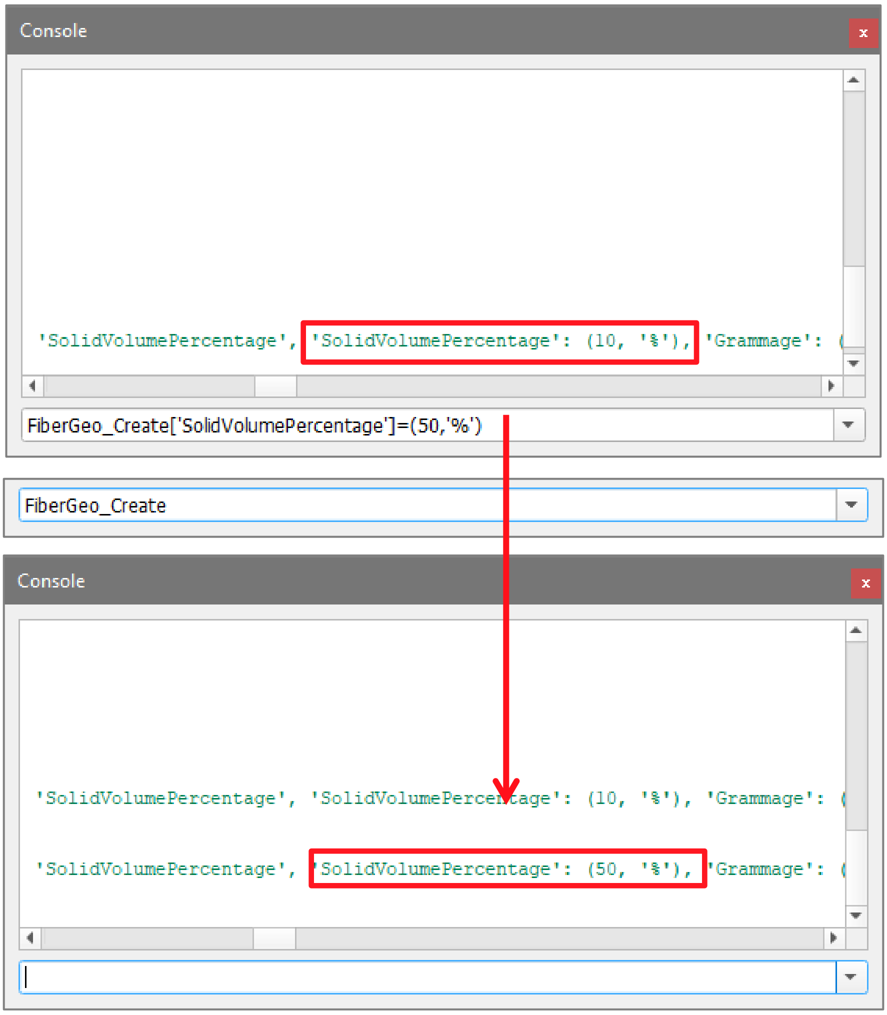
Besides, variables can be used. Store information in a variable for later use as, for example, the Python dictionary of the current FiberGeo parameters:



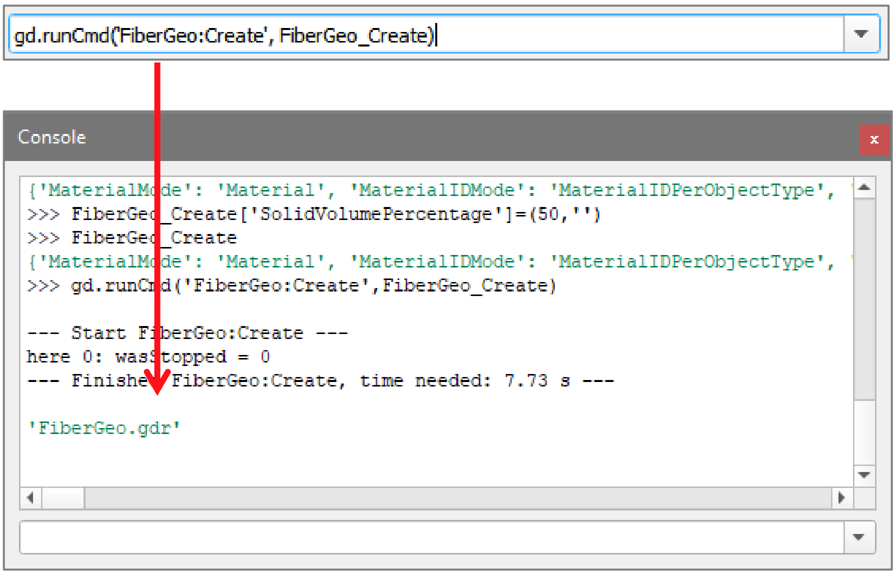
Typing the variable name again displays the value in the console. In the example, the Python dictionary from the FiberGeo **Create Options** dialog is shown.



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.

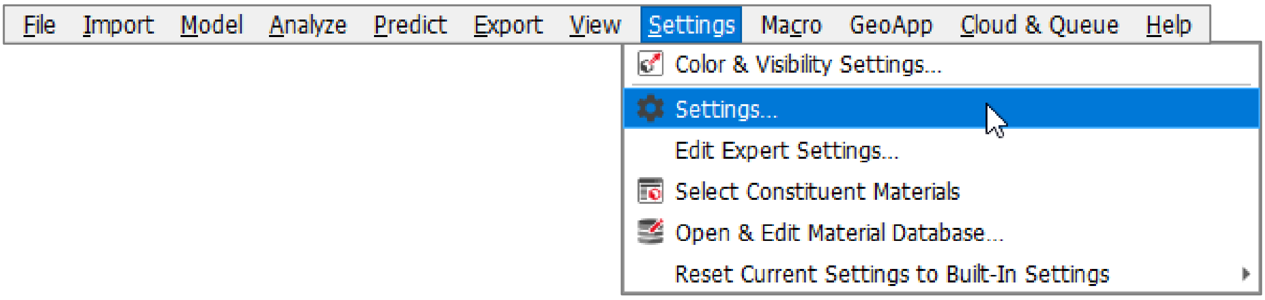


Now FiberGeo can be run with a solid volume percentage of 50 instead of 10, using the Python API command **gd.runCmd()** which is described on page 50.

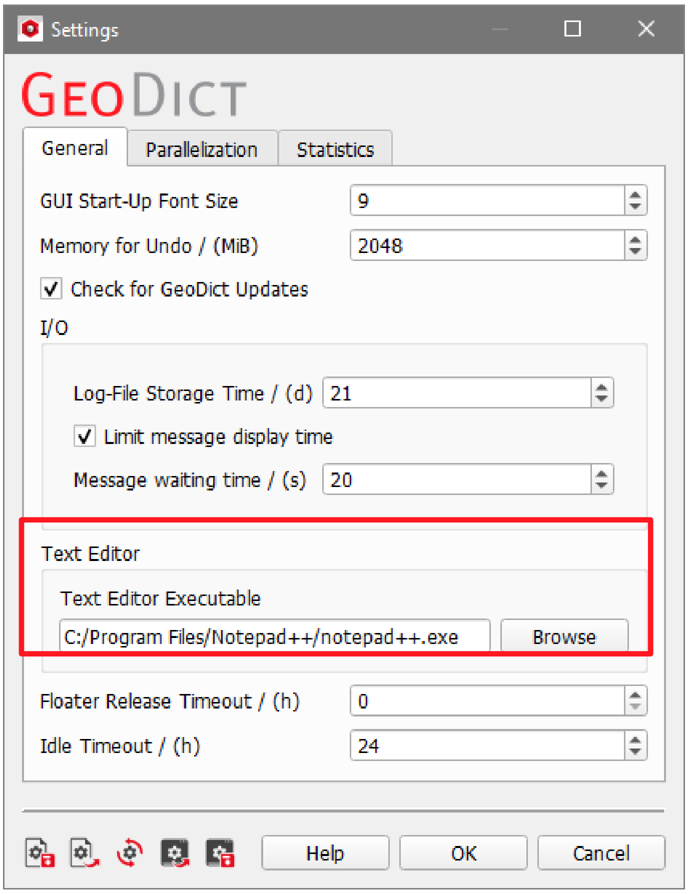


Choosing a text editor to edit a macro

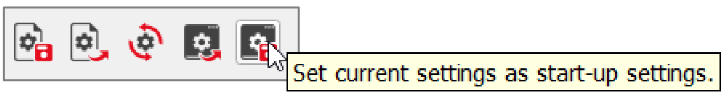
To define e.g. Wordpad as the default text editor, open GeoDict and select **Settings** → **Settings ...** from the menu bar.



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



To always open the macros in the selected editor, remember to store the settings as start-up settings by clicking the corresponding icon in the **Settings** dialog.



Click **OK** to apply the editor change.

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, enter the path to the desired editor.

EDITORS AVAILABLE FOR **WINDOWS** USERS

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. The user can also define his/her own syntax highlighting. **Notepad++** is also included in the GeoDict-Tools installer.

EDITORS AVAILABLE FOR **LINUX** USERS

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.

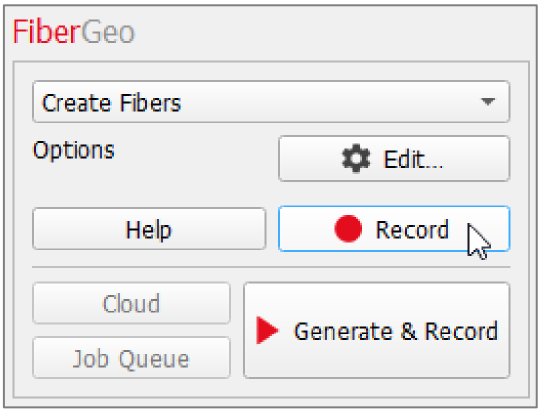
PARAMETER MACROS FOR PARAMETER STUDIES

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.

TRANSFORMING A SIMPLE MACRO INTO A PARAMETER MACRO FOR A PARAMETER STUDY

The user starts 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 as explained in page 7. We gave it the name "simple_macro.py". Then select **Module → FiberGeo** and click **Record**.



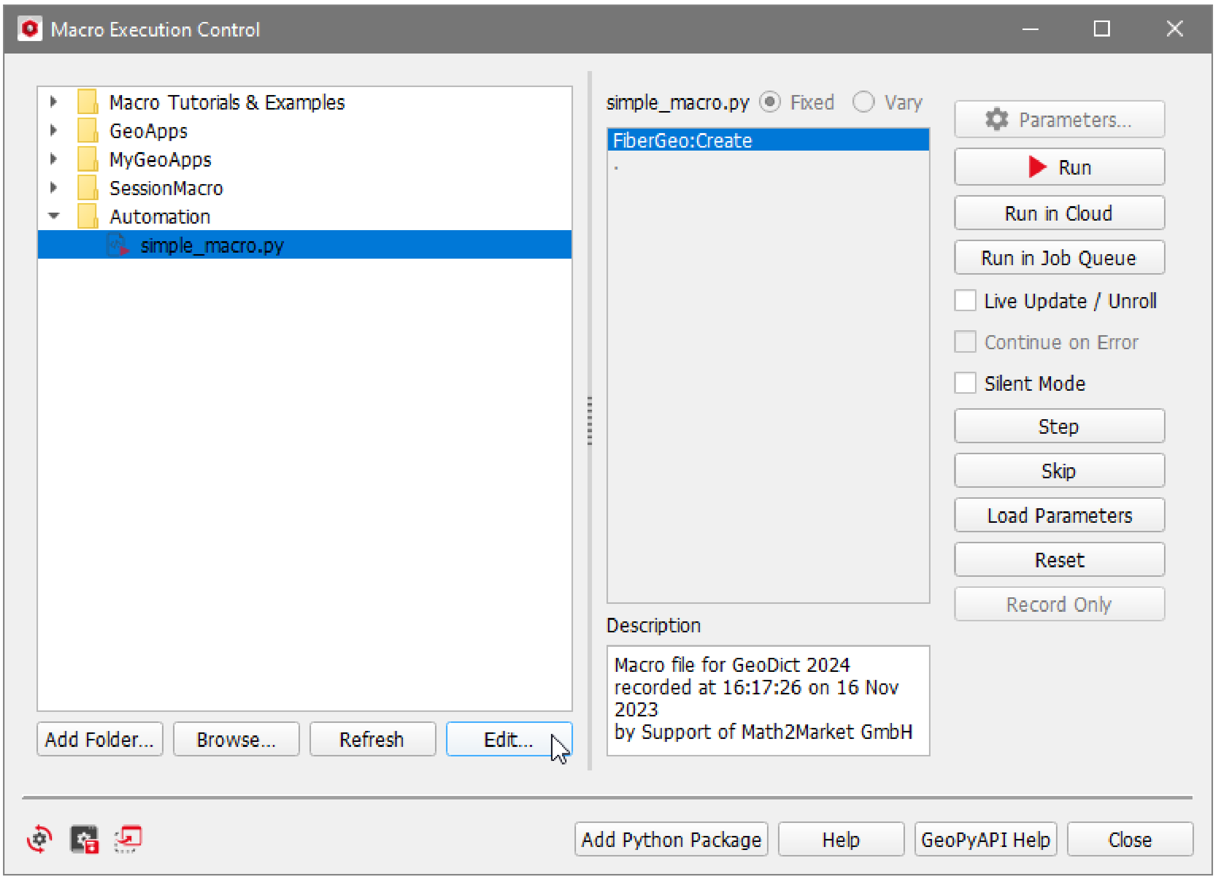
The single value for **Random Seed** is 42 and the single value for **Object Solid Volume Percentage** is 10.

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++).

```
Variables = {
  'NumberOfVariables' : 0,
#  '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'
#     },
  }
```

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.

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 this example.

EDITING THE MACRO

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,
RandomSeed and Fiber Diameter in combination to generate random seed series
of increasingly dense fibrous structures with infinite circular fibers.
'''

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
  },
}
```

In the **Variables** block, (as shown above) change the **NumberOfVariables** to **3** and un-comment the **Variable1** by deleting the **#** signs.

Use copy-paste to add a second and third variable element.

'Variable1' is given the Name **gd_SVP**, **'Variable2'** is given the Name **gd_RandomSeed** and **'Variable3'** 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') and their 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 to save the results in the new folder **'Variable_Study'**. Find out more details about the variables block on page 44.

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

In the block **FiberGeo:Create**, the **Domain** parameters are not modified

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).

```
   'MaximalTime'              : (24, 'h'),
   'OverlapMode'              : 'AllowOverlap',
   'NumberOfObjects'          : 100,
   'StoppingCriterion'        : 'SolidVolumePercentage',
   'SolidVolumePercentage'    : (gd_SVP, '%'),
   'Grammage'                 : (10, 'g/m^2'),
```

Right underneath of **Random Seed**, 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.

```
   'PercentageType'           : 0,
   'RandomSeed'               : gd_RandomSeed,
   'IsolationDistance'        : (0, 'm'),
   'ResultFileName'           : f'FiberGeo_{gd_SVP}_{gd_RandomSeed}_{gd_FiberDiameter}.gdr',
   'MatrixDensity'            : (0, 'g/cm^3'),
   'OverlapMaterial' : {
```

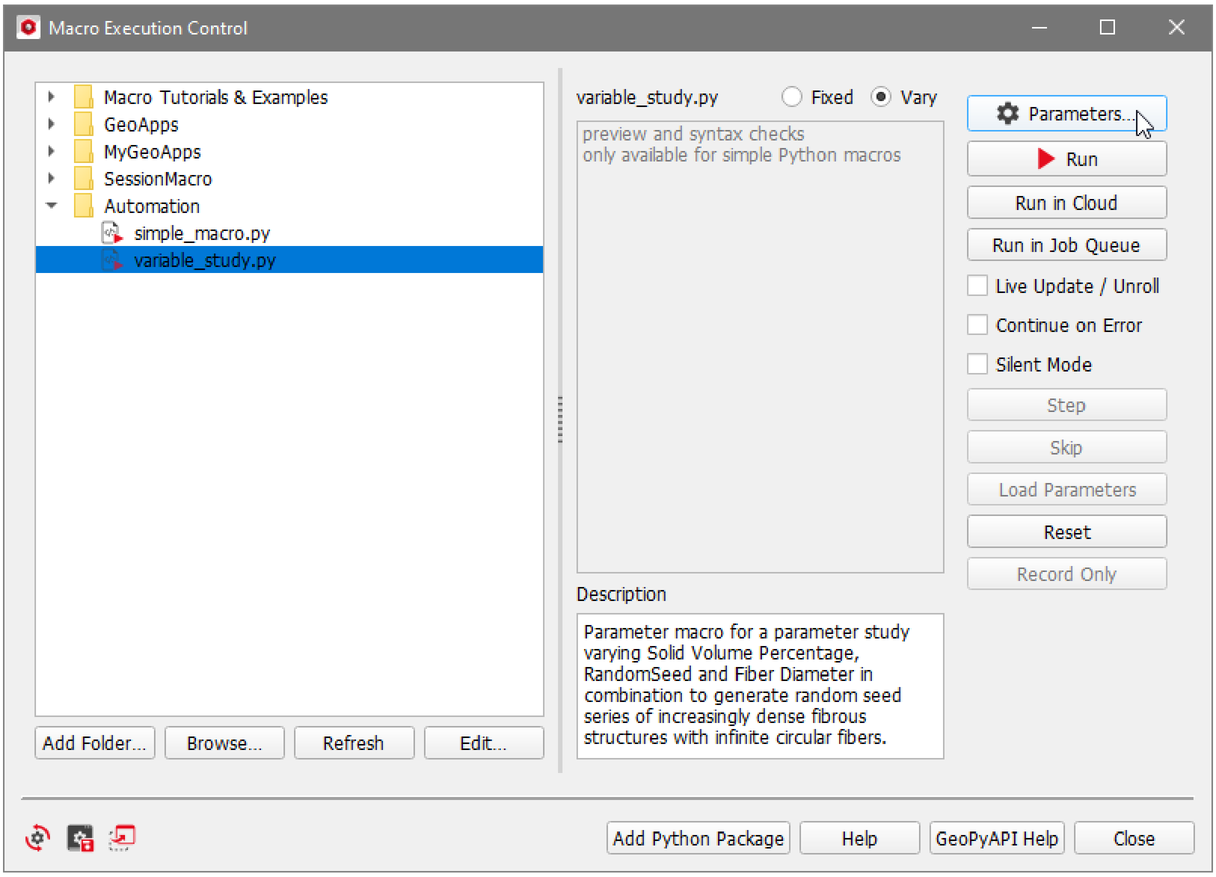
Finally, in the block **Generator1**, more precisely in the subblock **DiameterDistribution** replace the Value 1e-05 by **gd_FiberDiameter * 1e-06**. The factor 1e-06 is needed, as the fiber diameters in the dictionary must be given in meter. Thus, the fiber diameter of the first fiber type can be changed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

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

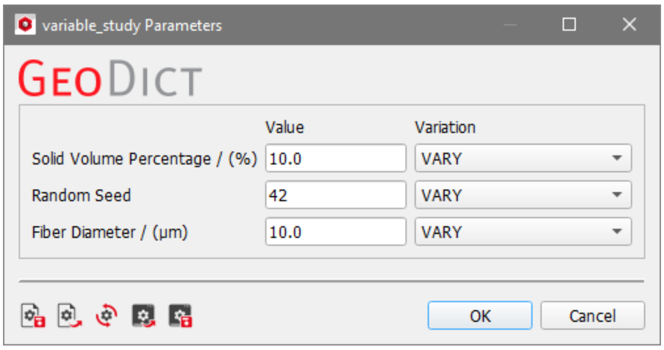
In the editor, save the modified macro as **variable_study.py** (NotePad++: **File → Save As...**)

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

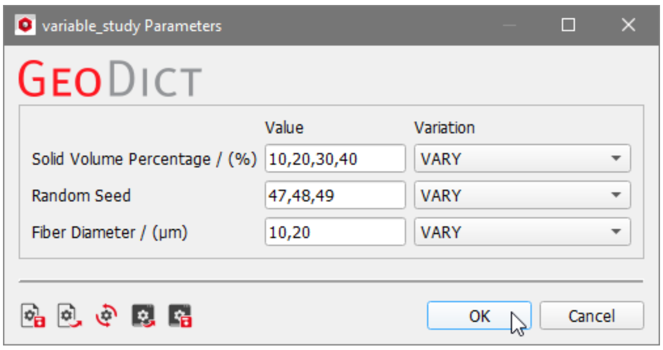
The text entered under **Description –** in the edited macro is shown in the description area and, since now the macro contains variables, **Vary** is available to be checked. Check it and click the **Parameters** button.



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.



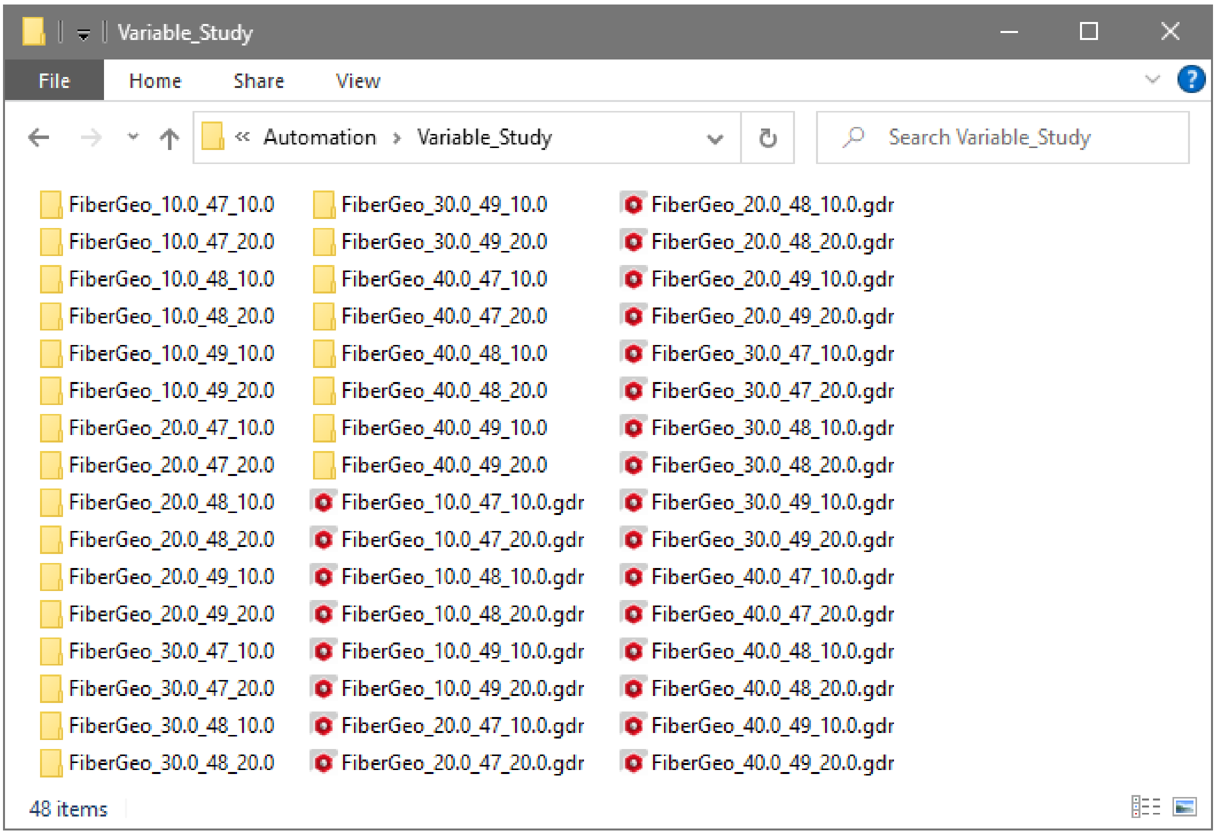
To set the parameter study, enter four values of increasing **Solid Volume Percentage** (10%, 20%, 30% and 40% SVP), three random seed values (e.g. 47, 48 and 49) and two values for **Fiber Diameter** (e.g. 10 and 20). Leave the **Variation** for all three at **VARY**.



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).



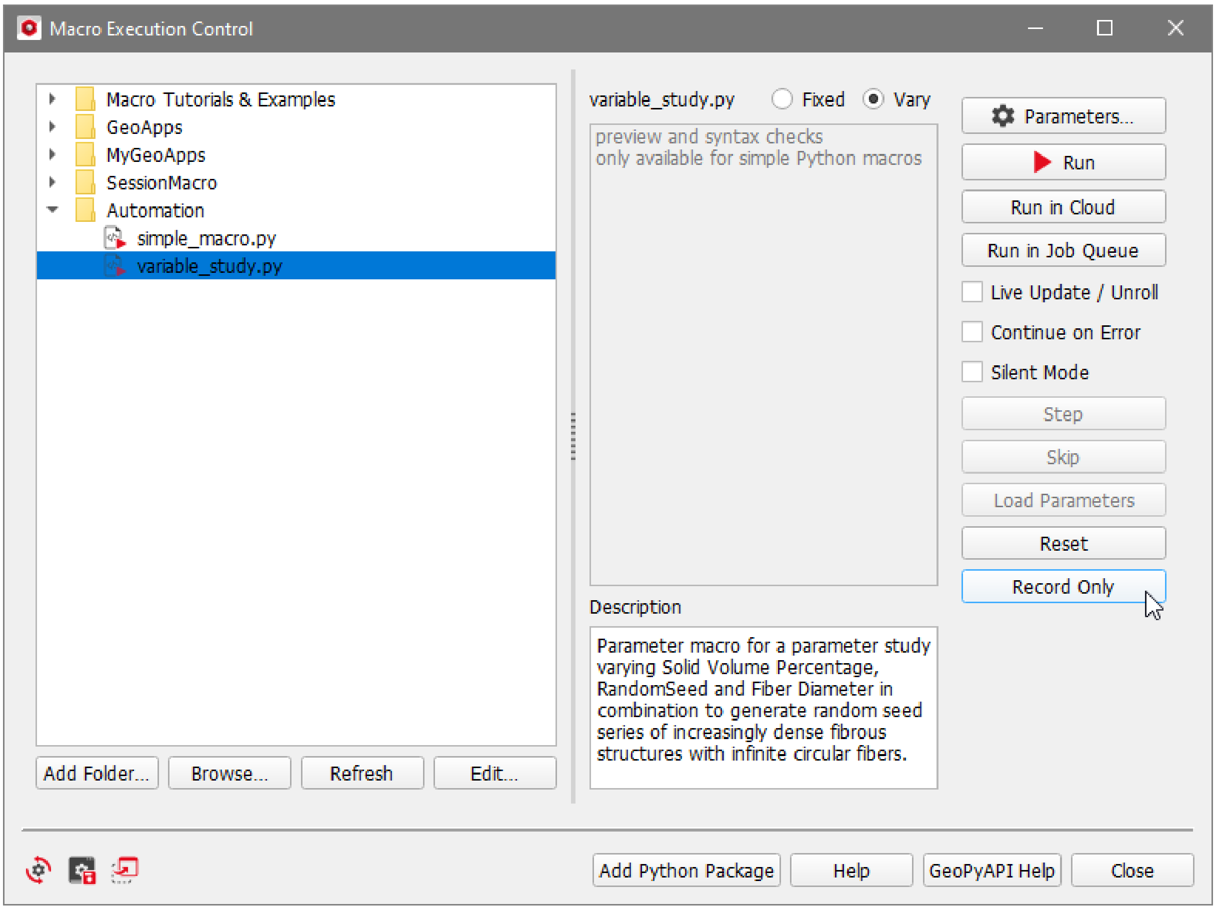
These 24 result files can be opened in GeoDict, and the Result Viewer offers the possibility to combine some or all results in a plot. See the Result Viewer User Guide for more details.

Starting VaryMacro from Python

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 page 7.

Open the **Macro Execution Control**, check **Vary** and edit the parameters for the variable study as desired (explained on pages 39ff).

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



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**.

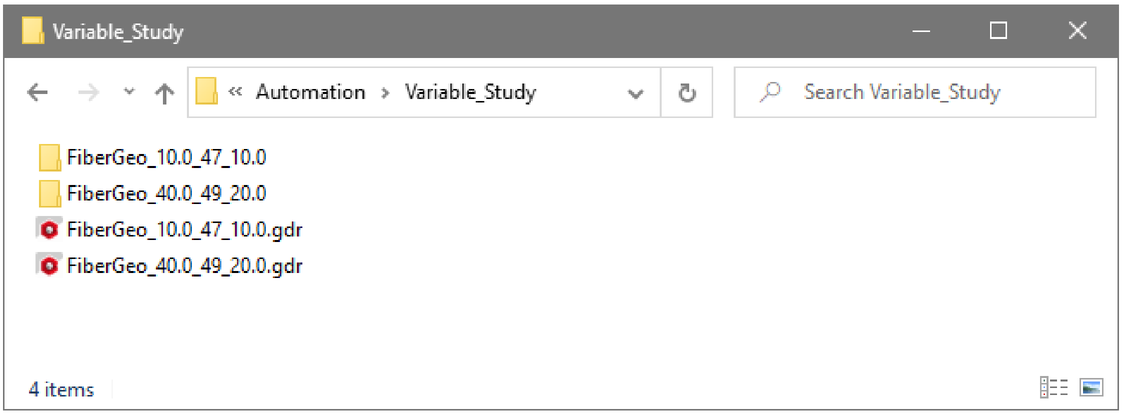
The **GeoDictMacro:VaryPython** command is located after the **Variables** section. This command can be used for any parameter macro. The file path and the variables have to be given. The entries in the Variables dictionary correspond to the vary parameters dialog box, described on pages 14ff.

```
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.



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 = {
   'NumberOfVariables' : 0,
#   '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'
#     },
   }

# 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
# Type:           mandatory, known types are bool, boolgroup, double, uint, int, string, filestring,
folderstring, material, combo, table, combogroup, labelgroup
# Unit:           optional,  appears only in GUI (not used to rescale any input parameters automatically)
#                            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** (see page 15).

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 syntax described on page 18.

uint

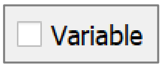
For a variable of type **'uint'** only nonnegative integer values are allowed for this variable, i.e. 0, 1, 2, … In the **Parameters** dialog it is also possible to change the value by clicking the arrows on the right or by turning the mouse wheel while the cursor is rested on the parameter box. If a 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 described on page 18.

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 described on page 18.

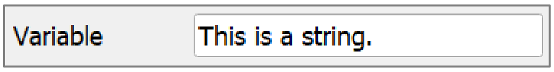
bool

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



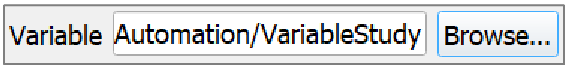
string

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



folderstring

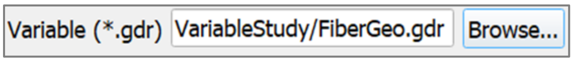
For a variable of type **'folderstring'** in the **Parameters** dialog a **Browse** button will appear next to the parameter box to search for the desired folder on the computer.



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'.

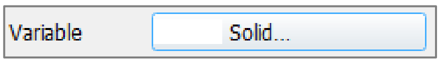
```
'Type'              : 'filestring',
'Unit'              : 'gdr',
```



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'.

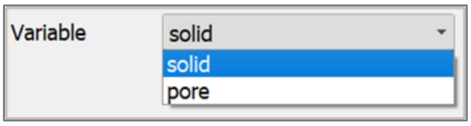
```
'Type'              : 'material',
'Unit'              : 'solid',
'BuiltinDefault'    : 'Manual'
```



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.

```
'Type'                  : 'combo',

'Unit'                  : 'solid;pore',
```

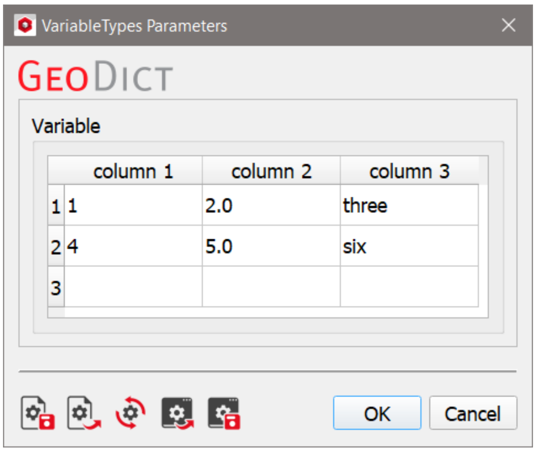


table

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.

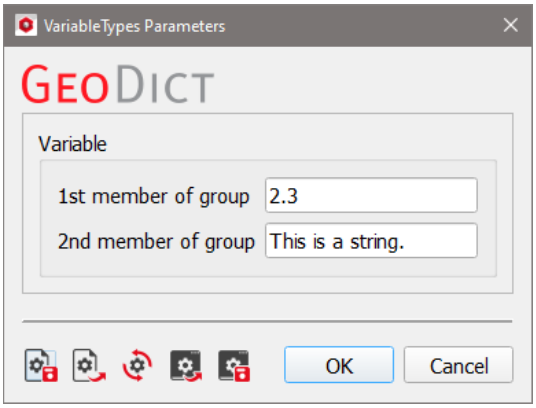



```
'Variable1' : {
   'Name'              : 'gd_table',
   'Label'             : 'Variable',
   'Type'              : 'table',
   'Unit'              : ['int','float','string'],
   'ColumnHeaders'     : ['column 1', 'column 2', 'column3'],
   'BuiltinDefault'    : [1, 2.0, 'three', 4, 5.0, 'six']
}
```

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'**.

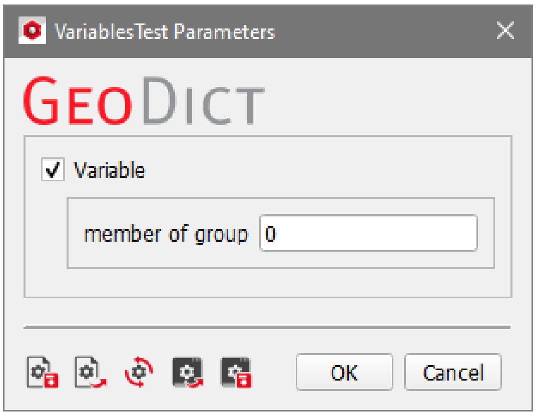



```
Variables = {
  'NumberOfVariables'     : 3,
    'Variable1' : {
      'Name'              : 'gd_labelgroup',
      'Label'             : 'Variable',
      'Type'              : 'labelgroup',
      'Member'            : ['member1', 'member2'],
      'BuiltinDefault'    : True
    },
    'Variable2' : {
      'Name'              : 'member1',
      'Label'             : '1st member of group',
      'Type'              : 'double',
    },
    'Variable3' : {
      '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'**.

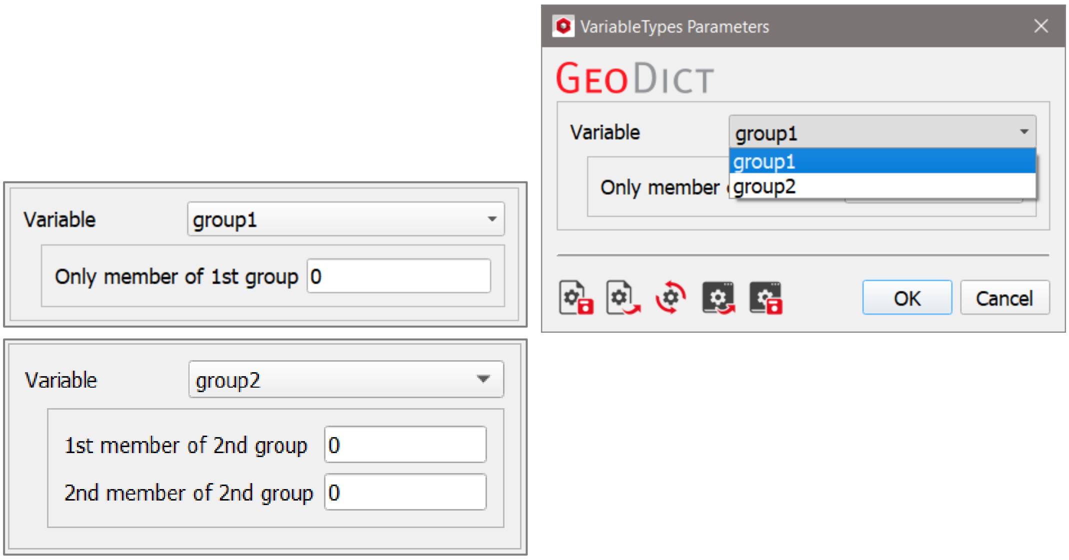


```
Variables = {
  'NumberOfVariables'     : 2,
    'Variable1' : {
      'Name'              : 'gd_boolgroup',
      'Label'             : 'Variable',
      'Type'              : 'boolgroup',
      'Member'            : {'true' : ['member']},
      'BuiltinDefault'    : True
    },
    'Variable2' : {
      '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.

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 = {
  'NumberOfVariables'     : 4,
  'Variable1' : {
    'Name'                : 'gd_combogroup',
    'Label'               : 'Variable',
    'Type'                : 'combogroup',
    'Unit'                : 'group1;group2',
    'Member'              : {'group1' : ['onlymember'],
                             'group2' : ['member1', 'member2']},
    'BuiltinDefault'      : True
    },
  'Variable2' : {
    'Name'                : 'onlymember',
```

```
    'Label'              : 'Only member of 1st group',
    'Type'               : 'double',
  },
  'Variable3' : {
    'Name'               : 'member1',
    'Label'              : '1st member of 2nd group',
    'Type'               : 'int',
  },
  'Variable4' : {
    'Name'               : 'member2',
    'Label'              : '2nd member of 2nd group',
    'Type'               : 'int',
  }}
```

Python Scripting in GeoDict

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.

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

In addition, a special object called **gd** is available everywhere within a Python macro. The whole GeoDict API (Application Programming Interface) is exposed via the **gd**-object.

GeoDict Application Programming Interface (API)

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.

General Functions

gd.runCmd(cmdName, args, versionString)

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

- cmdName is the name of the command as they appear in the **Session Macro** dialog described on page 24, e.g. "GeoDict:LoadFile" to load a GDT file.
- args is a python dictionary holding the arguments (see below)
- versionString is a string containing the GeoDict version for which this macro was written, e.g. "2024"

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.

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

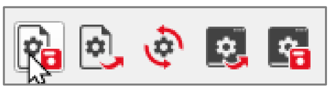
```
gd.runCmd("GeoDict:Terminate", {}, "2024")        # terminates GeoDict, the
                                                      dictionary for this command is
                                                      empty
```

gd.runCmdIgnoreExtraKeys(cmdName, args, versionString)

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

gd.runCmdFromGPS(gps_file_path)

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

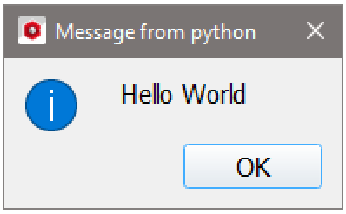


```
gd.runCmdFromGPS("FiberGeo.gps")        # generates a fiber structure from
                                          a *.gps file
```

GD.MSGBOX(BASIC PYTHON VALUE)

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. Example:

```
gd.msgBox("Hello World")
```



GD.SHOWGDR(PATH)

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

GD.GETBLOCKER()

Get a Blocker object to prevent GeoDict Dialogs from showing up. Use this function via 'with' keyword. 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", {'FileName' :   # loads a GeoDict structure file
    f'example{i}.gdt'})
    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 save an image
    SaveThreeDImage_args)
```

GD.GETDOMAIN(VERSIONSTRING)

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.

```
domain = gd.getDomain('2024')           # 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']['VoxelLength']                # assigns the voxel length to
                                                      variable vl

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

print(f'The loaded structure has {nx} voxels in     # show message in the console
     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 follows:

```
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. Example:

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

GD.GETVOXELCOUNTS2D(DIRECTION:INT, MATERIAL INDEX : INT)

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. Example:

```
voxelCounts = gd.getVoxelCounts2D(2,1)              # get the number of voxels 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} voxels have       # show message dialog
     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. Example:

```
nx , ny, nz = gd.getVolDimensions()                 # get the number of 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 and assign it to
                                                      variable ID_1
```

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

GD.GETSOLIDVOLUMEFRACTION()

Returns the solid volume fraction of the currently loaded geometry.

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

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

GD.GETVIEWSTATUS(VERSIONSTRING)

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, e.g. to change the angle of the camera:

```
d = gd.getViewStatus("2024")                     # get the current rendering settings
d["Camera"]["Camera3D"]["Rotation"]=[38,22,-65]  # change angle of camera
gd.runCmd("GeoDict:SetViewStatus", d, "2024")    # update settings
```

GD.GET2DVIEWASPLOT(INT DIRECTION, INT SLICE, BOOL ORIENTATION)

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). How to create a custom result file is explained on page 84. Input the desired view direction, slice and if the image orientation should be **Top to Bottom** (True) or **Bottom to Top** (False). The view direction must be given as integer, where 0 = X, 1 = Y and 2 = Z.

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

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

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

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

resultfile.write()                               # write result file
```

GD.GETBUILTINDEFAULTS(STRING COMMANDNAME)

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

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

```
Create_args['SolidVolumePercentage'] = 20          # change solid volume fraction to
                                                     20%
gd.runCmd("FiberGeo:Create", Create_args)          # version is omitted - defaults to
                                                     latest
```

GD.GETCURRENTSETTINGS(STRING COMMANDNAME)

Returns the current settings argument dictionary for a command. This can then be modified and passed to **runCmd**. Example:

```
Create_args =                                      # get the arguments for
    gd.getCurrentSettings("FiberGeo:Create")         "FiberGeo:Create"
Create_args['SolidVolumePercentage']=(20, '%')     # change solid volume fraction to
                                                     20%
gd.runCmd("FiberGeo:Create", Create_args)          # version is omitted - defaults to
                                                     latest
```

GD.SETCURRENTSETTINGS(STRING COMMANDNAME, PARAMETERS DICTIONARY, VERSION STRING)

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

```
Create_args =                                      # get the arguments for
    gd.getCurrentSettings("FiberGeo:Create")         "FiberGeo:Create"
Create_args['SolidVolumePercentage']=(20, '%')     # change solid volume fraction to
                                                     20%
gd.setCurrentSettings("FiberGeo:Create",           # set the changed settings in the
    Create_args, "2024")                             FiberGeo Options dialog
```

GD.SETTEMPERATURE(TEMPERATURE FLOAT, UNIT STRING)

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

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

GD.GETCONSTITUENTMATERIALS()

Returns the map of the current constituent materials as Python dictionary. 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              # show message dialog of result
    {ID_0_Type}.")
```

GD.GETDATABASEMATERIAL(STRING NAME)

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

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

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

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

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

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

gd.msgBox(f"At  {air_temp}  degrees  {air_temp_u}  the   # show message dialog
    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.

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

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

foldercontent = os.listdir(databasefolder)        # get the content of the folder 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.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.

```
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    contains    GAD  # show message dialog
    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 doesn't contain GAD   # show message dialog
    objects.")
```

GD.GETNUMBEROFGADOBJECTS()

Returns the number of loaded GAD objects as an integer. Example:

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

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

GD.GETGADOBJECT(INT ID, VERSIONSTRING)

Returns the settings of the GAD object with the given index id (first object has id 1) as a Python dictionary. For an example see **getSelectedGADObjects()** below.

GD.GETSELECTEDGADOBJECTS()

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

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

```
GAD_Selection = gd.getSelectedGADObjects()      # get IDs of selected gad objects
GAD_ID = GAD_Selection[0]                       # choose smallest selected GAD
                                                  object ID
GAD_Object = gd.getGADObject(GAD_ID," 2024")    # get the settings of the
                                                  corresponding gad object
gd.msgBox(GAD_Object['Type'])                   # show type of selected GAD_object
                                                  in message box, e.g. sphere,
                                                  ellipsoid, circular fiber, …
```

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.

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 voxel is located  # shows message box
    at position {Voxels[0]}")
```

GD.GETSETTINGSFOLDER()

Returns the settings folder as a string.

Windows: c:\user\%USERNAME%\GeoDict2024

Linux: ~/.geodict2024

```
SettingsFolder = gd.getSettingsFolder()         # assigns the file path of the
                                                  settings folder to the variable
                                                  SettingsFolder
gd.msgBox(f"The GeoDict settings can be found in # shows the settings folder in a
    \n {SettingsFolder}")                          message box
```

GD.GETINSTALLATIONFOLDER()

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

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

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

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() instead.

GD.GETGDFOLDER()

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

GD.GETGDGEOAPPSFOLDER()

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

GD.GETGDVIDEOMACROFOLDER()

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

GD.GETMACROFILEFOLDER()

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

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

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

GD.GETMACROFILENAME()

Returns the macro name as a string. Example:

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

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

GD.GETPROJECTFOLDER()

Returns the current project folder of GeoDict as a string. Example:

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

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

GD.GETHOSTNAME()

Returns the name of the host as a string. Example:

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

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

GD.GETSTANDARDFILEHEADER()

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

```
Header = gd.getStandardFileHeader()    # assigns the string of the 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. Example:

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

gd.msgBox ( f"The current GeoDict version is   # show message dialog
    {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. Example:

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

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

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). 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      # transform all entries of the row
          voxel_value in row]                in strings and write them in the
                                             string list strList
```

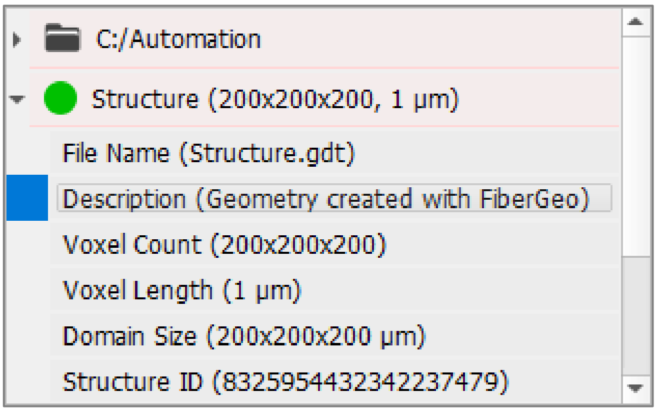
```
    fd.write(",".join(strList) + "\n")          # writes all entries of strList in
                                                  the csv file, separated by
                                                  commas, adds a new line at the
                                                  end of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

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

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.



For an example, see underneath the **getStructureHash64** command.

GD.GETSTRUCTUREHASH()

This function is deprecated, use gd.getStructureHash64() 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()**. Example:

```
import stringmap                              # imports the Python module
                                                stringmap

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

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

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

```
if int(GDR_Hash_64) == Structure_Hash_64:        # condition: if the IDs are equal,
                                                     the following indented section
                                                     is executed
  gd.msgBox("The loaded structure belongs to the # show message dialog
    result file FiberGeo.gdr")
else:                                            # if the IDs are not equal, the
                                                     following indented section is
                                                     executed
  gd.msgBox("The loaded structure does not belong # show message dialog
    to the result file FiberGeo.gdr.")
```

GD.GETSTRUCTUREFILENAME()

Returns the structure file name of the currently loaded structure as a string. Example:

```
filename = gd.getStructureFileName()             # assigns the currently loaded
                                                     structure's file name to the
                                                     variable filename
gd.msgBox(f"Currently {filename} is loaded.")    # show message dialog
```

GD.GETNUMBEROFTRIANGLES()

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

```
Number = gd.getNumberOfTriangles()               # assigns the number of triangles to
                                                     the variable Number
gd.msgBox ( f"The number of triangles is         # show message dialog
    {Number}")
```

GD.GETTRIANGULATIONBOUNDINGBOX()

Returns the bounding box of the current triangulation. If no triangulation exists {{0,0,0}, {0,0,0}} is returned. 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-  # show the result in a message
    direction.")                                     dialog
```

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

GD.GETVOLUMEFIELD(INDEX OR NAME)

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** can be used to obtain the desired information. 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 = gd.getVolumeFieldsInfo()          # get list of dictionaries of 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 assigning
                                                 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 = gd.getVolumeField(Name)          # assign the numpy array 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 values
    VolumeField[i][j][k]                         of the kth Z-layer to the sum
                                                 value_sum

        value_count = value_count + 1          # count the summands of value_sum
  meanVal = value_sum / value_count            # compute mean value of all pore
                                                 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 layers", Name)  # add a graph object with the 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, Name)       # add a single dataset with the 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.GETPROGRESS(STR TEXT, INT STEPS, STR SPLASH, BOOL GRAPH, BOOL HAS STOP BUTTON)

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.

The input parameters are:

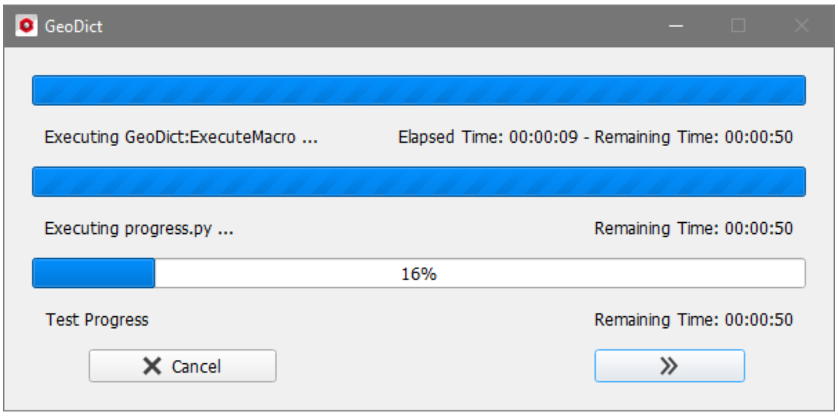
- Progress bar name as a string (obligatory)
- Total number of steps as an integer value (obligatory)
- Splash screen as a string. Displays the given splash screen in the progress dialog. Entering a random string displays the default GeoDict splash screen. Omit parameter or enter an empty string ('') to obtain a progress dialog without a splash screen.
- Add a graph to progress dialog if True is entered. No graph is displayed if the parameter is omitted or set to False.
- 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.

The progress bar has the following functions:

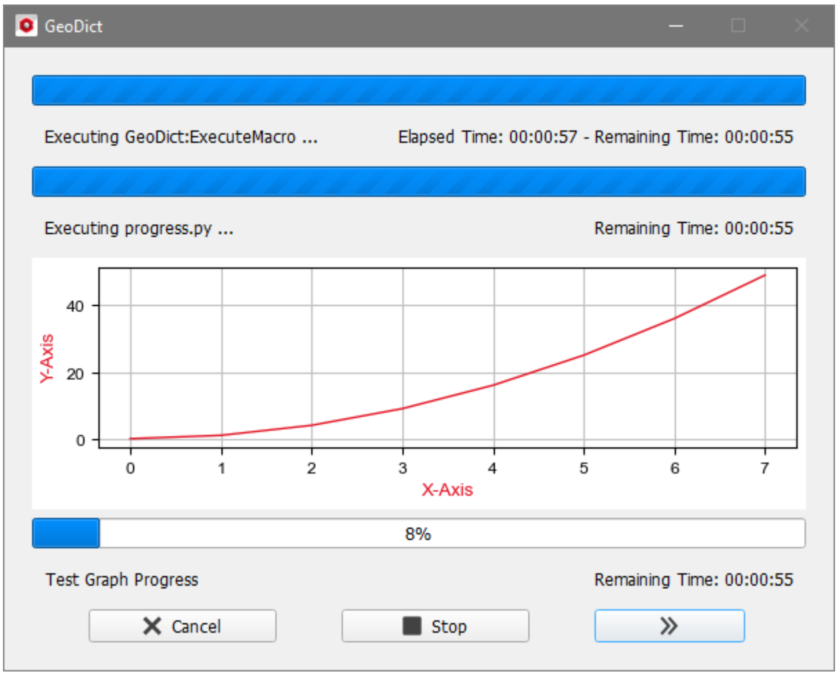
- update(int step) updates the progress bar to the specified step.
- updateWithGraph(int step, str X-axis label, X-value, Y-axis label, Y-value) updates the progress bar to the specified step and also displays and updates a graph with the given values
- wasCancelled() checks if the cancel button was hit.
- wasStopped() checks if the stop button was hit.

Example:

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

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

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

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

del progress                                      # delete the progress bar
```




```
progress2 = gd.getProgress("Test Graph      # create a second progress bar
    Progress", 80, '', True, True)              about 80 steps that is named
                                                "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

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

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

del progress2                                 # delete the progress bar
```



GD.SETSTRUCTURE(3D NUMPY ARRAY, FLOAT VOXEL LENGTH)

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. For example, if a 3D structure is saved as a *.csv file, structured in the same way as in the example for **gd.getStructure** 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 fd:          # open input file for reading and
                                                  assign it to fd. The file is
                                                  closed after the last indented
                                                  line following

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

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

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

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

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

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

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

  Structure = voxel_values.reshape(nx, ny, nz)  # reshape the 1-dimensional 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.SETSTRUCTUREDESCRIPTION(STRING DESCRIPTION)

Sets the description text for the currently loaded structure.

Example:

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

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

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

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

GD.UPDATEGEOMETRY()

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

GD.UPDATEVOLUMEFIELD(STRING PATH)

This command has no return value but updates the visualization of a volume field.

GD.MAKEDIALOG(STRING KEY, STRING TITLE)

Creates an input dialog object to query the user for parameters. It is used as follows:

- Create a dialog object: gd.makeDialog(key, title)
  - **key** 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** is an optional argument giving the window title of the dialog.

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

- Add (input) fields to the dialog, e.g.:

```
dlg.addBoolInput("myBooleanParameter", "This is a checkbox", init=True,
    tooltip="This is a tooltip")
# The returned value is "True" if the checkbox is checked and "False" if not

dlg.addIntegerInput("myIntegerParameter", "This is an integer input", min=5, max=10,
    init=6, tooltip="This is a tooltip")
# The returned value is the inserted integer

dlg.addUintegerInput("MyUintInput", "This is an uinteger input",  min = -5, max =
    5, init=0, tooltip="Choose an integer parameter within the boundaries")

dlg.addFloatInput("myFloatParameter", "This is a float input", min = -3.5, max =
    5.2, init=2.1, tooltip="This is a tooltip")

dlg.addTextInput("myStringParameter", "This is a free form text input box",
    init="This is a String", tooltip="This is a tooltip")

dlg.addFileInput("myFileSelection", "This allows you to browse for files having a
    given extension", "gdt", init="File.gdt", tooltip="This is a tooltip")

dlg.addFolderInput("MyFolderInput", "This allows you to browse for a folder")

dlg.addComboInput("myComboBox", "A combobox to select one of a list of items",
    ["first item", "second item", "third item"], tooltip= "This is a tooltip")
# The returned value is the index of the selected item, e.g. 0 for the first item,
    1 for the second etc.

dlg.addComboInputString("ComboString", "A combobox to select one item from a list",
    ["first item", "second item", "third item"])
# The returned value is the string of the selected item

dlg.addMaterialInput("MyMaterialInput", "This allows you to choose a material from
    the material data base")
dlg.addTableInput("MyTableInput", "This is a table input.", types = "int,float",
    columnHeaders=["left","right"], init=[[1,2.0],[3,4.0]])
```

  - These arguments are optional keyword arguments:
    - the **init** argument gives the initial value for the field (the built-in default).

- **tooltip** specifies a description string that is shown when the user hovers the mouse over the input field.
- **min/max** arguments restrict the range of input (only available for integer, uinteger and float input).

- It is also possible to write e.g.:

```
dlg.addIntegerInput("myNewIntegerParameter", "This is an integer parameter without limits but with a default value", init=42, tooltip="Enter some value here.")
```

- Free-form text can also be added using this function:

```
dlg.addText("Any arbitrary text", 20, True)        # add bold text to the dialog with font size 20
dlg.addText("More arbitrary text", 10, False)      # add text to the dialog with font size 10 and not bold
```

- Fields can be grouped within a box as follows:

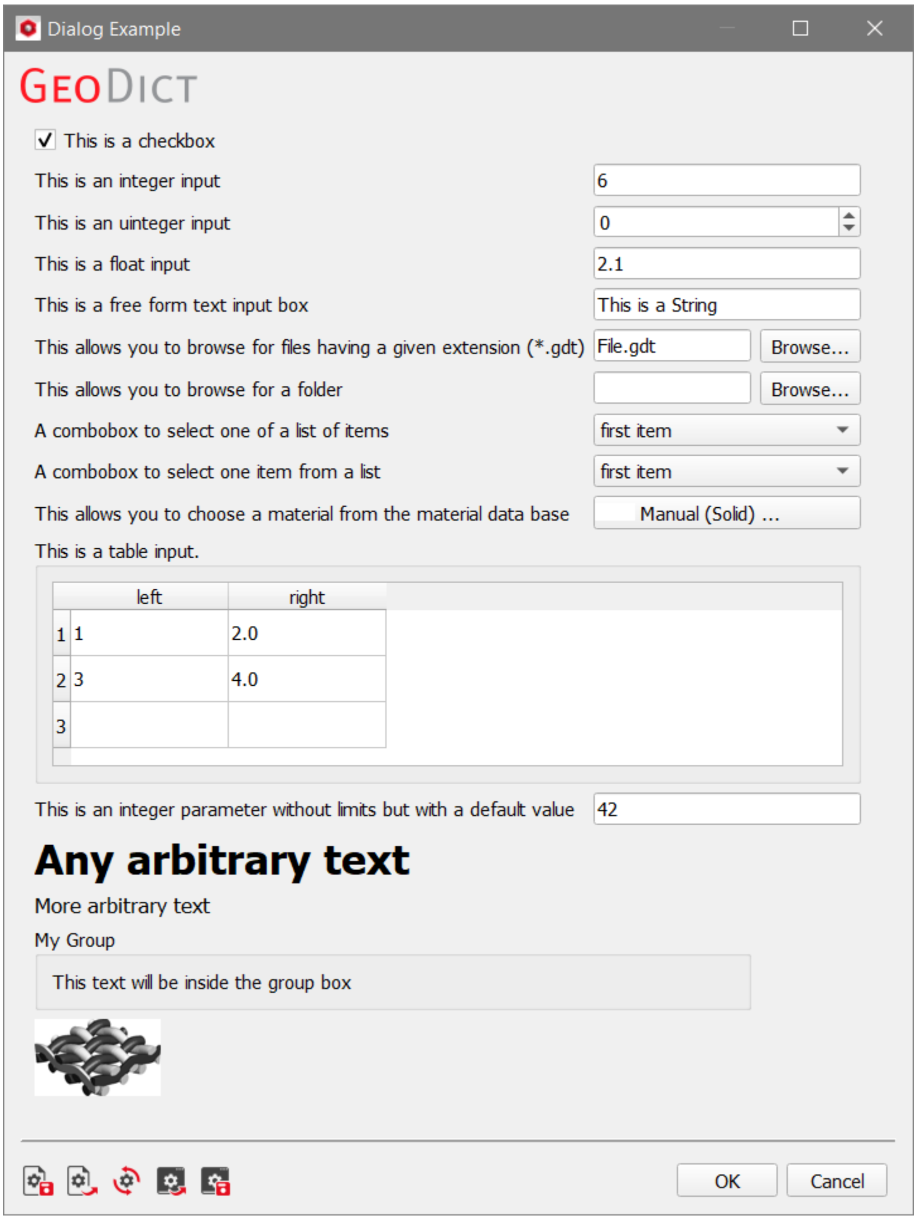
```
dlg.beginGroup("My Group")                          # start the group for the box
dlg.addText("This text will be inside the group box")   # add content (text, input fields, …)
dlg.endGroup()                                      # end the group
```

- Furthermore, images can be added to the dialog box as 3D numpy arrays

```
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 = image.resize((100,round(100*h/w))) # resize image to fit in the dialog, without changing aspect ratio
I = np.asarray(image)                      # transform image to a 3D numpy array
dlg.addImage(I)                            # add image to the dialog
```

- Execute the dialog:

```
result = dlg.run()
```



- If the user clicks **Cancel**, result will be **None**.
- Otherwise, result will be a dictionary containing the entered values, e.g.

```
gd.msgBox("The user has selected the file:" + result["myFileSelection"])
```

- Save the dialog settings as a *.gps file after calling run(). Define the desired file path as a string:

```
dlg.saveSettings("Example/MyFirstCustomDialogSettings.gps")
```

GD.MAKEGRAPHDIALOG()

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

- Create a graph dialog object:

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

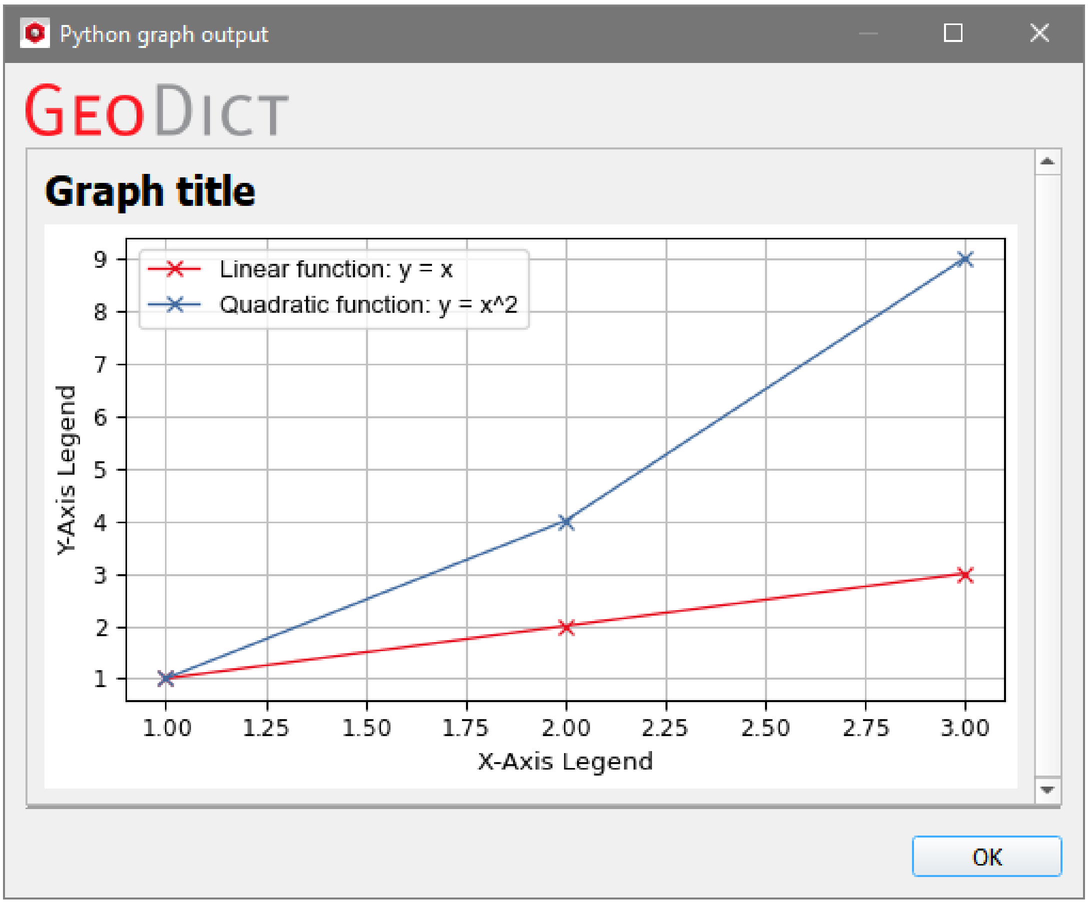
- Add graph input:

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

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

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

- Display the graph dialog:

```
gDlg.run()
```

When calling **gDlg.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.



ImportGeo-Vol specific Functions

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.

gd.ImportGeoVol.getHistogram()

Returns the histogram of the currently loaded image as a python list of tuples containing value and count each. 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 = gd.ImportGeoVol.getHistogram()    # get list of tuples describing 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.

gd.ImportGeoVol.getNewImageDimensions (direction)

Returns the current gray value image size in voxels in the desired direction, given as integer (0 for X-direction, 1 for Y-direction, 2 for Z-direction).

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

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

gd.ImportGeoVol.getNewImageResized(nx,ny,nz, bool is16Bit)

Creates a new 3D gray value image with the entered dimensions. If is16Bit (True or False) is not given 8 bits are used. Only used in ImportGeo custom python image filters, not in regular macros.

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.

GD.IMPORTGEOVOL.GETOTSUTHRESHOLD()

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

```
OTSU = gd.ImportGeoVol.getOtsuThreshold()          # get threshold and assign it to
                                                     variable OTSU
gd.msgBox(f"OTSU threshold is {OTSU}")             # show message box
```

GD.IMPORTGEOVOL.GETMULTIOTSUTHRESHOLD()

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

```
OTSU = gd.ImportGeoVol.getMultiOtsuThreshold()     # get threshold list and assign it
                                                     to variable OTSU
gd.msgBox(f"The OTSU thresholds are {OTSU}")       # show message box
```

GD.IMPORTGEOVOL.GETVOXELLENGTH()

Returns the currently in ImportGeo-Vol set voxel length. For an example, see below under the **setVoxelLength()** command.

GD.IMPORTGEOVOL.SETVOXELLENGTH(VOXEL LENGTH)

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

```
vl_old = gd.ImportGeoVol.getVoxelLength()          # get voxel length of currently
                                                     loaded gray value image and
                                                     assign it to variable vl_old
gd.ImportGeoVol.setVoxelLength(1e-6)               # set voxel length of currently
                                                     loaded gray value image to 1µm
vl_new = gd.ImportGeoVol.getVoxelLength()          # assign new voxel length to
                                                     variable vl_new
gd.msgBox(f"The voxel length was changed from      # show message box
    {vl_old} to {vl_new}")
```

GD.IMPORTGEOVOL.GETROTATIONSUGGESTION(FULL IMAGE, THRESHOLD)

The command returns the rotation suggested for the loaded gray value image. Therefore, it takes a bool (True or False) if full image should be suggested. If the parameter is set to “False”, plane is suggested. The parameter for threshold must be an integer. Example:

```
Rot = gd.ImportGeoVol.getRotationSuggestion(False)    # get rotation suggestion for
                                                        suggest plane and assign
                                                        it to variable rotation.
                                                        Threshold is deprecated -
```

```
                                                   default to automatic
                                                   threshold
Rotation_args =                                  # get Built-in Defaults for the
    gd.getBuiltinDefaults("ImageProcessing:Rotation")   Python dictionary 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:Rotation", Rotation_args)   # rotate the gray value image,
                                                   version is omitted –
                                                   default to latest
```

FilterDict Particle specific Functions

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

gd.getParticlesInfo()

Returns a Python dictionary containing the number of batches and the maximal and minimal batch animation times. 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 {Num}.")  # show message dialog
```

gd.getParticles(versionString)

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. For an example, see below in the other particle commands.

.getBatchInfo(int batch index)

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** command.

Example:

```
particles = gd.getParticles("2024")             # assigns the particles object to
                                                    the variable particles

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

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

.getDiameter(int batch index, int particle index, float time)

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

```
particles = gd.getParticles("2024")             # assigns the particles object to
                                                    the variable particles

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

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

.GETDIAMETERS(INT BATCH INDEX, INT PARTICLE INDEX, FLOAT TIME STEP)

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.

.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. Example:

```
particles = gd.getParticles("2024")          # assigns the particles object to
                                                 the variable particles

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

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

.GETMULTIPLICITIES(INT BATCH INDEX, INT PARTICLE INDEX, FLOAT TIME STEP)

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. Example:

```
particles = gd.getParticles("2024")          # assigns the particles object to
                                                 the variable particles

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

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

.GETMULTIPLICITY(INT BATCH INDEX, INT PARTICLE INDEX, FLOAT TIME)

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

```
particles = gd.getParticles("2024")          # assigns the particles object to
                                                 the variable particles

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

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

.GETPARTICLEINFO(INT BATCH INDEX, INT PARTICLE INDEX)

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

```
particles = gd.getParticles("2024")              # assigns the particles object to
                                                    the variable particles

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

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

.GETPOSITION(INT BATCH INDEX, INT PARTICLE INDEX, FLOAT TIME)

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

```
particles = gd.getParticles("2024")              # assigns the particles object to
                                                    the variable particles

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

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

.GETPOSITIONS(INT BATCH INDEX, INT PARTICLE INDEX, FLOAT TIME STEP)

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. Example:

```
particles = gd.getParticles("2024")              # assigns the particles object to
                                                    the variable particles
Pos = particles.getPositions(1, 3500, 0.0001)    # assign the list of tuples (time,
                                                    position) of a particle to the
                                                    variable Pos

print(f"The position of the particle is          # prints the second (time, position)
    {Pos[1][1]} at time {Pos[1][0]}.")              tuple to the console
```

.GETPOSITIONSATTIME(INT BATCH INDEX, FLOAT TIME)

Computes all (interpolated) particle positions at a given time.

```
particles = gd.getParticles("2024")              # assigns the particles object to
                                                    the variable particles

Pos = particles.getPositionsAtTime(1, 0.0001)    # assign the list of positions of
                                                    all particle at the given time
                                                    to the variable Pos

print(f"The position of the particles are        # prints positions to the console
    {Pos}.")
```

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. How to use the stringmap module is shown on page 81 and how use the gdr module is described on page 84.

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**.

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.
PIL (pillow)	Library to read, write, and manipulate images.
xlsxwriter	Create Excel files from GeoPy.
pptx	Library to create PowerPoint slides. Note: GeoDict provides a simplified wrapper API in the gd_ppt namespace, as described on page 76.
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.

If a module is needed which is not shipped with GeoDict, it can be installed as described on page 22.

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 an own template, the user saves a copy of his/her own corporate design PowerPoint template, containing only master slides. In PowerPoint, the user changes 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**.

The **gd_ppt** library is loaded at the beginning of a Python file with the command **import gd_ppt** and contains the following commands:

gd_ppt.ReportGenerator(template file)

Opens the template PowerPoint file.

add_slide(layout master index)

Adds a slide with the style defined by the Layout Master with the given index.

SAVE(FILE NAME)

Saves the PowerPoint presentation under the given name.

ADD_TEXT(PLACEHOLDER, TEXT, FONT_SIZE)

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. Example:

```
import gd_ppt                                        # import PowerPoint API wrapper
                                                        library

rep =                                                # create a report generator based
 gd_ppt.ReportGenerator("example_template.pptx")        on master slides in
                                                        "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.",   # fill the placeholder text with
     font_size = 45)                                    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**.

ADD_PICTURE(PLACEHOLDER, PICTURE FILE)

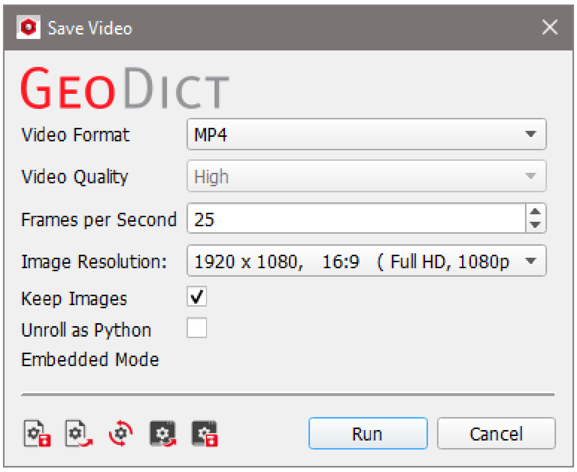
Fills a picture in the given picture placeholder. For this command, a ***picture placeholder*** is needed. Example:

```
import gd_ppt                                        # import PowerPoint API wrapper
                                                        library

rep =                                                # create a report generator based
 gd_ppt.ReportGenerator("example_template.pptx")        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.png from
                                                            the project folder

rep.save("report_example.pptx")                         # save the PowerPoint presentation
                                                            with the name
                                                            report_example.pptx
```

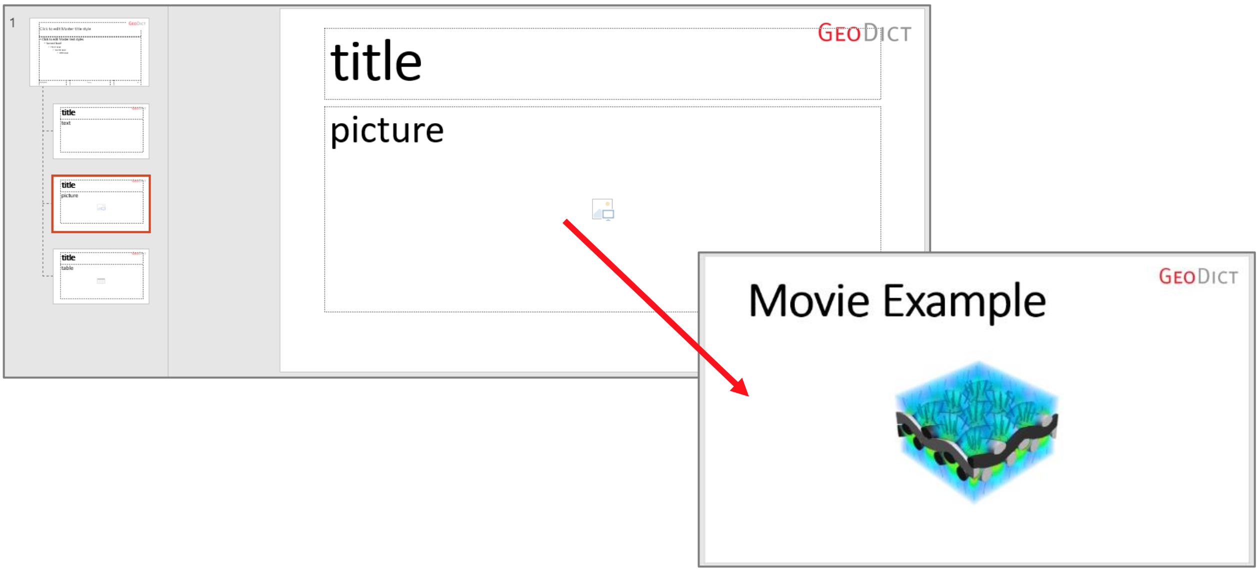
ADD_MOVIE(PLACEHOLDER, MOVIE FILE)

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 name **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.



Example:

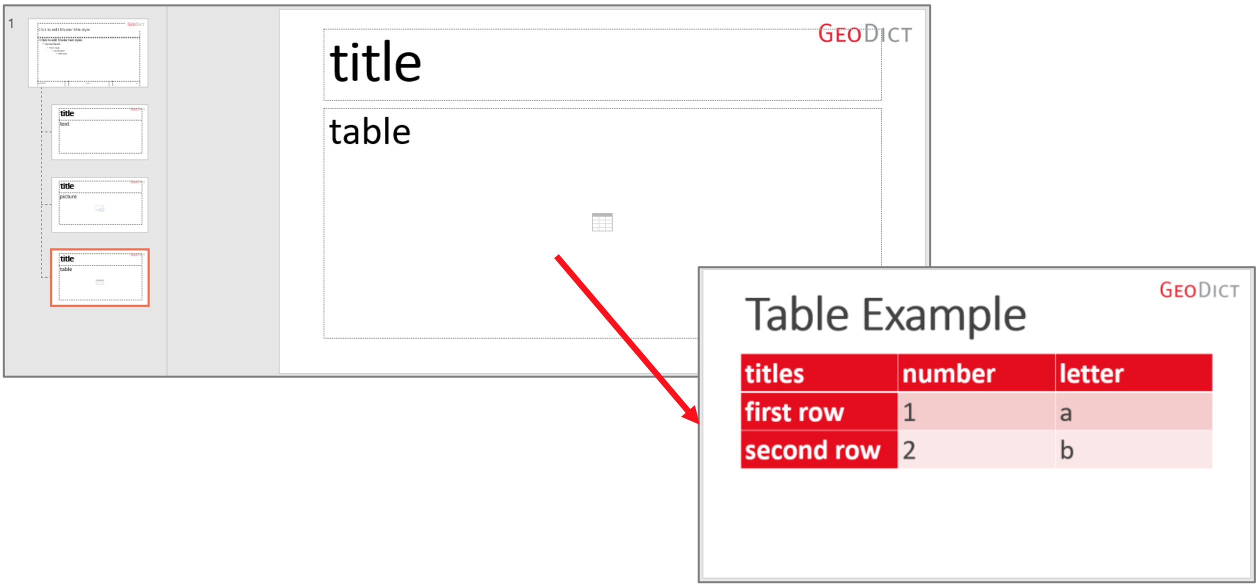
```
import gd_ppt                                           # import PowerPoint API wrapper
                                                            library

rep =                                                   # create a report generator based
 gd_ppt.ReportGenerator("example_template.pptx")            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", "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.mp4 from the
                                                            project folder

rep.save("report_example.pptx")                         # save the PowerPoint presentation
                                                            with the name
                                                            report_example.pptx
```



ADD_TABLE(PLACEHOLDER, TABLE, HORIZONTAL_HEADER, VERTICAL_HEADER, FONT_SIZE)

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. Example:

```
import gd_ppt                                   # import PowerPoint API wrapper
                                                  library

rep =                                           # create a report generator based
 gd_ppt.ReportGenerator("example_template.pptx")  on master slides in
                                                  "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 Example")           # fill the placeholder title 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", "second row"]     # assign a list for the 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, horizontal_header = # fill the placeholder table with
    h_h, vertical_header = v_h, font_size = 50)   the defined table and the
                                                  headers h_h and v_h, and font
                                                  size 50

rep.save("report_example.pptx")                 # save the PowerPoint presentation
                                                  with the name
                                                  report_example.pptx
```



In the examples above, 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()**.

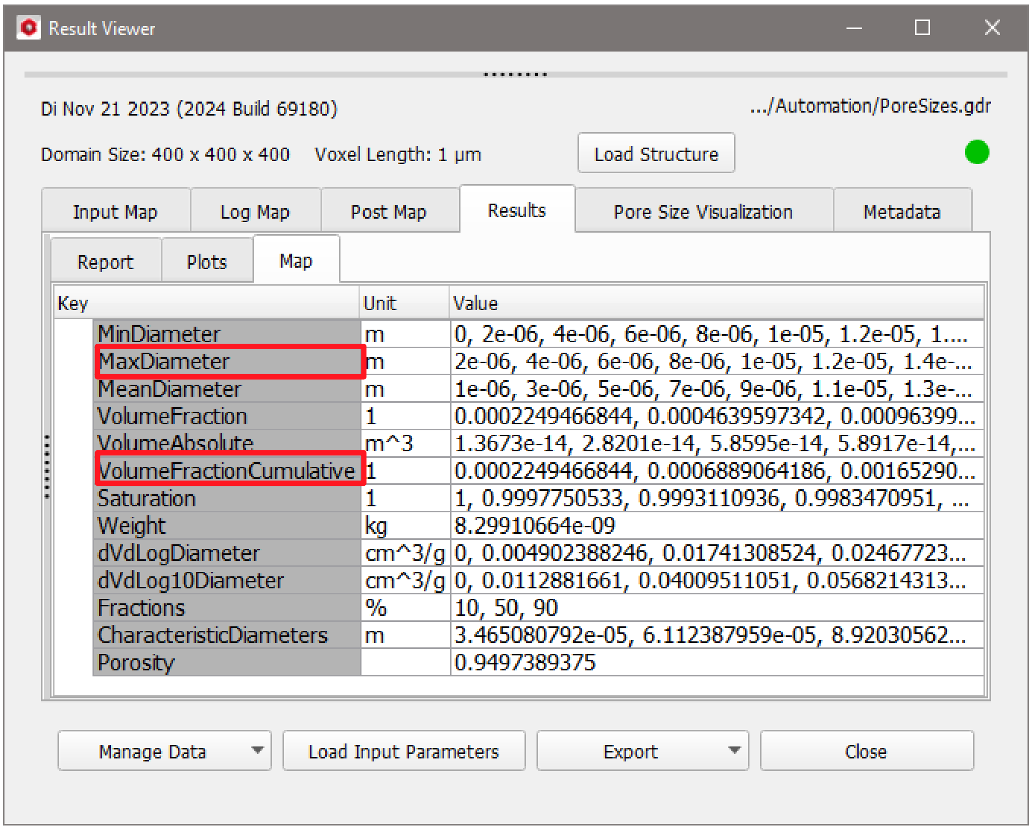
Access to GeoDict result files (*.gdr)

Accessing GeoDict result files 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.

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"
# a pore size distribution with PoroDict has to be run first to obtain the *.gdr file

gdr = stringmap.parseGDR(gdrPath)
# read and parse the GDR file into a string map object called "gdr"

gdr.push("ResultMap")
# make all further operations work on the subtree called "ResultMap"

# get the list values called "MaxDiameter" and "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 = gdr.getList("MaxDiameter")
# alternatively, you can omit the push before and write "ResultMap:MaxDiameter" here
volFracsCumulative = gdr.getList("VolumeFractionCumulative")

# do the following to convert the string lists to numerical values
maxDiameters = [float(x) for x in maxDiameters]
# convert each list entry from a string to a floating point value
volFracsCumulative = [float(x) for x in volFracsCumulative]
# convert each list entry from a string to a floating point value

gdr.pop()
# go back to the root of the tree
```

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(GDRPATH STRING)

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 ":".

```
import stringmap                              # import the stringmap module

gdrPath = "C:/Automation/PoreSizes.gdr"       # define the file path of the 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(KEY STRING)

All operations following this command only consider the given subtree.

.POP()

After push was used, this command leads back to the root of the tree.

.GETUNIT(KEY STRING)

Returns the unit for the given key as a string.

```
import stringmap                              # import the stringmap module

gdrPath = "C:/Automation/PoreSizes.gdr"       # define the file path of the gdr
                                                  file to consider"

gdr = stringmap.parseGDR(gdrPath)             # parse the gdr file
weight_Unit = gdr.getUnit("ResultMap:Weight") # get the unit for weight
weight = gdr.getDouble("ResultMap:Weight")    # get the value for weight as
                                                  number
```

```
print(weight, weight_unit)                  # print the resulting values
```

.GETDOUBLE(KEY STRING)

Returns the value for the given key as floating point number. For an example, see above.

.GETINT(KEY STRING)

Returns the value for the given key as integer number.

.GETBOOL(KEY STRING)

Returns the value for the given key as bool (True or False).

.GETLIST(KEY STRING)

Returns the value for the given key as list.

.HASKEY(KEY STRING)

Checks if the given key is contained in the considered tree and returns True or False accordingly. For an example see below for getFullKey.

.GETFULLKEY(KEY STRING)

Returns the full key path for the given key, e.g. ResultMap:Porosity, if Porosity is given for a pore size distribution result.

```
import stringmap                            # import the stringmap module
gdrPath = "C:/Automation/PoreSizes.gdr"     # define the file path of the gdr
                                               file to consider"
gdr = stringmap.parseGDR(gdrPath)           # parse the gdr file

if gdr.hasKey("Weight"):                    # if the key "Weight" is contained
                                               in the gdr the following
                                               indented section is executed

  key_path = gdr.getFullKey("Weight")       # get the full key 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 the GeoDict
  print ("The key is invalid."):               console
```

.TOFILE(FILE PATH STRING)

Saves the stringmap to the given file. 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 = "C:/Automation/PoreSizes.gdr"     # define the file path of the gdr
                                               file to consider"
gdr = stringmap.parseGDR(gdrPath)           # parse the gdr file

gdr["ResultMap:Weight"] = [0.0025, "mg"]    # set the weight to 0.0025 mg
                                               instead of 2.5e-10 kg.
gdr.toFile("NewResultFile.gdr")             # save the new map under a new
                                               name"
```

CREATE CUSTOM GEODICT RESULT FILES (*.GDR)

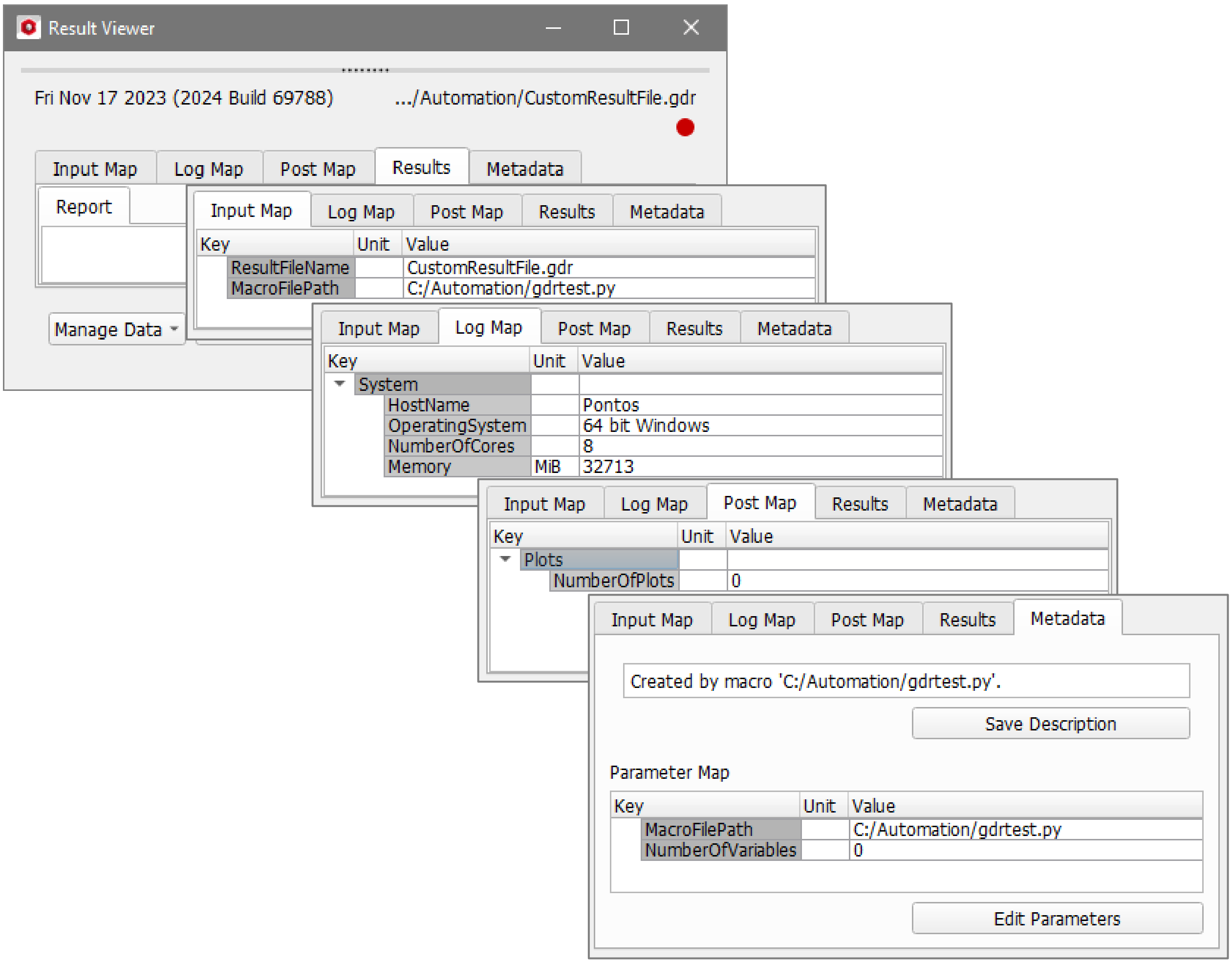
GeoDict includes an API to create custom result files (*.gdr). This is particularly useful, if the same workflow is repeated often with different parameters in an automatic parameter study and the results should be presented in the GeoDict **Result Viewer**. In this way, the library **gdr** provides 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 **import gdr** and contains the following commands:

GDR.GDR.CREATEEMPTYRESULS(GDR FILE NAME, RELEASE)

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. Example:

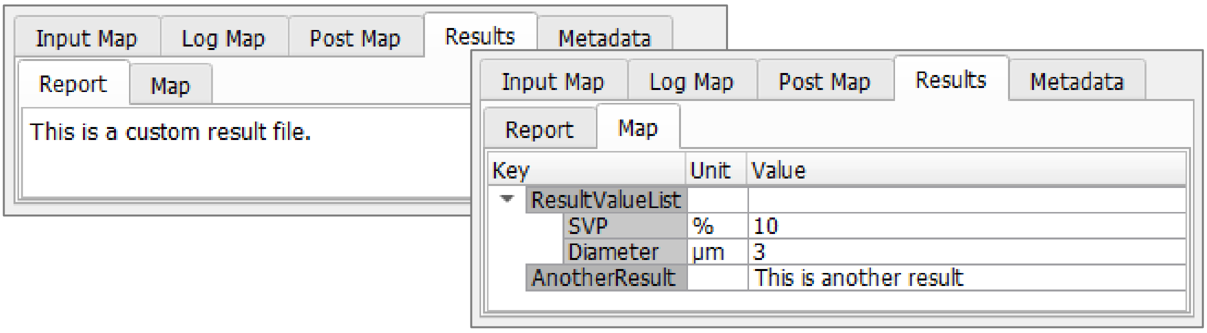
```
import gdr                                            # import gdr library

gdrf =                                                # open a result file with the
 gdr.GDR.createEmptyResults("CustomResultFile","2024")    name
                                                          CustomResultFile.gdr and
                                                          assign it to the
                                                          variable gdrf (GeoDict
                                                          result file)
```



SAVERESULTS(RESULTMAP, REPORT, RELEASE)

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() and saveResults(). Example:

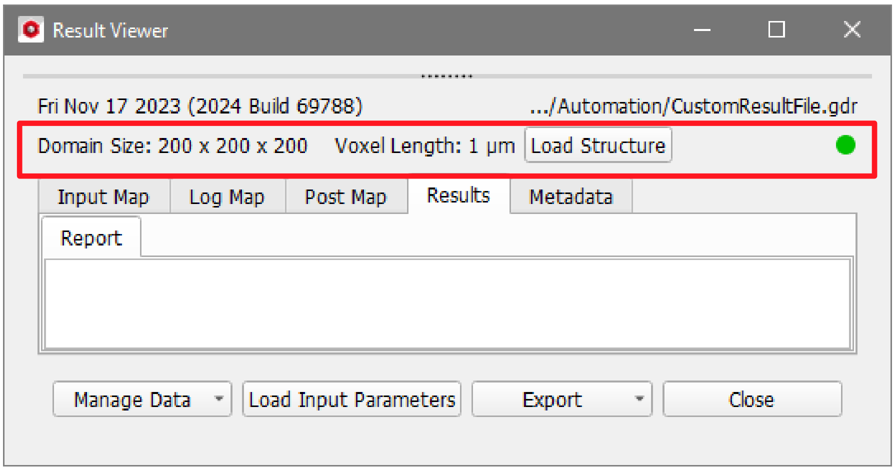
```
import gdr                                           # import gdr library

gdrf =                                               # open a result file with the
 gdr.GDR.createEmptyResults("CustomResultFile","2024")    name
                                                         CustomResultFile.gdr and
                                                         assign it to the
                                                         variable gdrf

report = "This a custom result file."                # define text for the Report
                                                         tab and assign it to
                                                         variable report

resultMap = {                                        # define content for the map
  'ResultValueList' : {                                  subtab of the Results
    'SVP'       : (10, '%'),                             tab in the result file
    'Diameter' : (3, 'µm')},                             and assign it to
  'AnotherResult' : 'This is another result'}            variable resultMap

gdrf.saveResults(resultMap, report, "2024")          # write and save the file
                                                         CustomResultFile.gdr
                                                         with report and Result
                                                         map
                                                     # result file is
                                                         automatically opened in
                                                         Result Viewer
```



SAVEGEOMETRY(FILENAME, RELEASE)

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. Example:

```
import gdr                                               # import gdr library

gdrf =                                                   # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")       the name
                                                             CustomResultFile.gdr
                                                             and assign it to the
                                                             variable gdrf
gdrf.saveGeometry("2024")                                # define text for the
                                                             Report tab and
                                                             assign it to
                                                             variable report
gdrf.saveResults({},"", "2024")                          # write and save the file
                                                             CustomResultFile.gdr
                                                             with empty report
                                                             and empty Result map
```



GEOMETRYMAP = PYTHON DICTIONARY

For advanced users. Adds explicitly given **Geometry** data to the generated result file. In most cases it is recommended to use the addGeometry() 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.

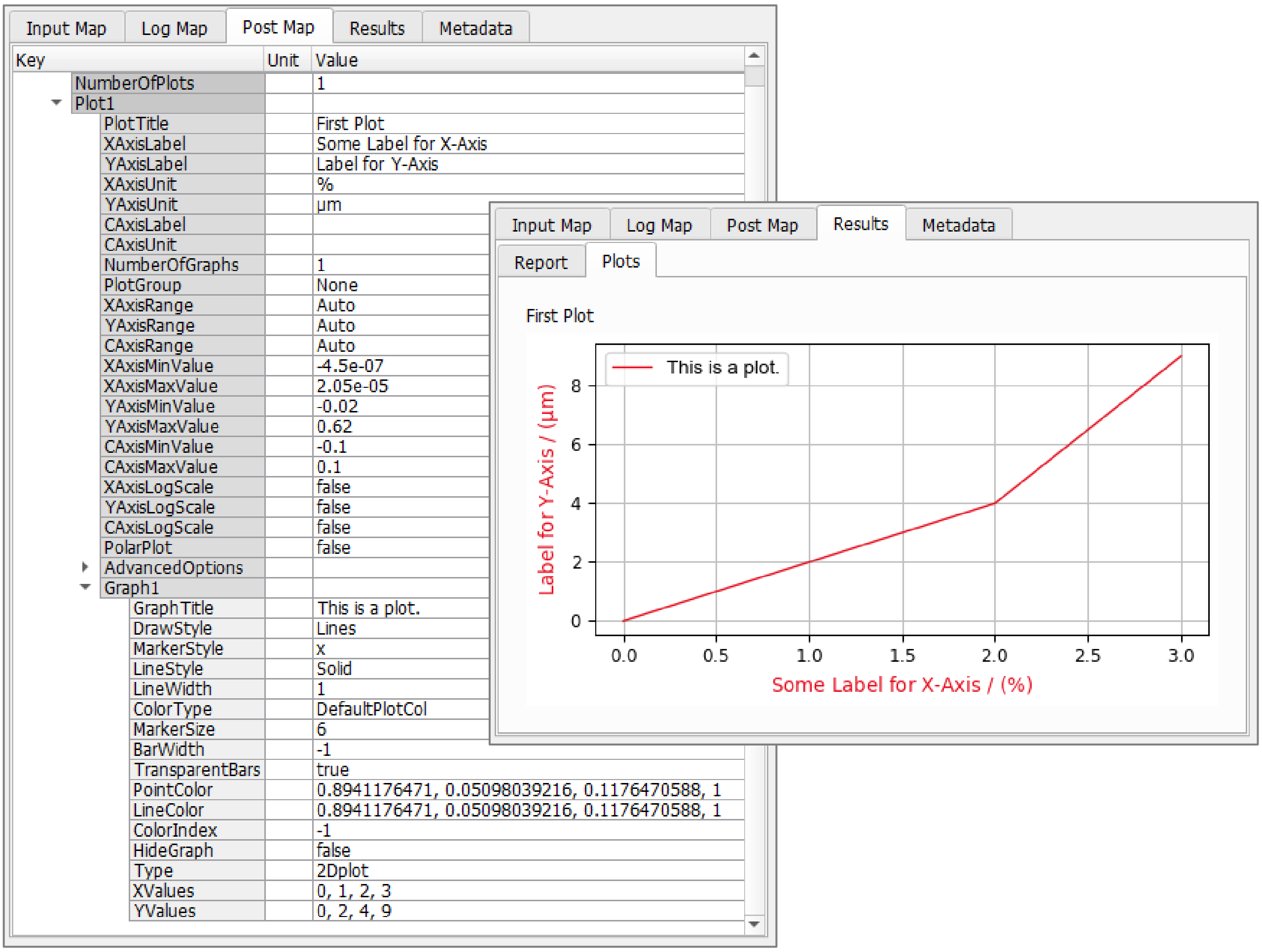
```
import gdr                                                  # import gdr library

strucHash      = gd.getStructureHash()                      # get structure hash
strucHash64    = gd.getStructureHash64()                    # get structure hash 64
strucDesc      = gd.getStructureDescription()               # get structure name
nx,ny,nz       = gd.getVolDimensions()                      # get number of voxels in
                                                              x-,y- and z-
                                                              direction

voxelLength    = gd.getVoxelLength()                        # get voxel length in
                                                              meter
volDimension   = nx*ny*nz                                   # compute total number of
                                                              voxels
svp            = gd.getSolidVolumeFraction()                # for a structure with
                                                              two solid materials
                                                              assigned to material
                                                              ID 1 and ID 2, this
                                                              computes the solid
                                                              volume percentage


gdrf =                                                      # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")        the name
                                                              CustomResultFile.gdr


GeometryParameters = {                                      # assign a Python
  'Hash'                 : strucHash,                         dictionary
  'Hash64'               : strucHash64,                       containing geometry
  'FileName'             : 'Example.gdt',                     data to the variable
  'NX'                   : nx,                                GeometryParameters
  'NY'                   : ny,
  'NZ'                   : nz,
  'UseBoxels'            : False,
  'VoxelLength'          : (voxelLength,'m'),
  'SolidVolumeFraction'  : svp}

gdrf.geometryMap = GeometryParameters                       # add geometry data to
                                                              result file
```

```
gdrf.saveResults({},"", "2024")        # write and save the file
                                         CustomResultFile.gdr
                                         with empty report
                                         and empty Result map
```

ADDPLOT(PLOT TITLE, X LABEL, Y LABEL, X UNIT, Y UNIT, X VALUES, Y VALUES, GRAPH TITLE)

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. Example:

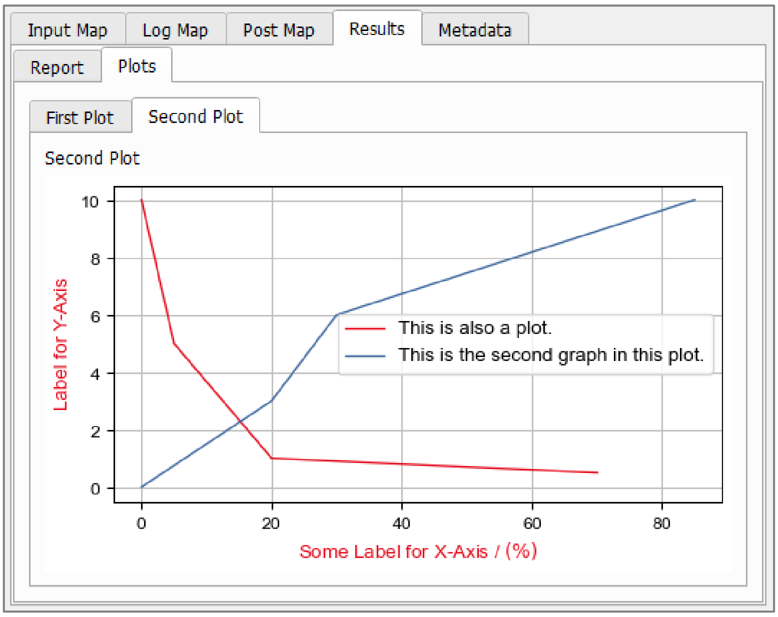
```
import gdr                              # import gdr library

gdrf =                                  # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")
                                          the name
                                          CustomResultFile.gdr
                                          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 list
GraphName = "This is a plot."           # define a graph name
gdrf.addPlot(Plotname, XLabel, YLabel, '%', 'µm', XValues,   # add plot to result file
    YValues, GraphName)                   with name, labels,
                                          units, values and
                                          graph name

gdrf.saveResults({},"", "2024")         # write and save the file
                                          CustomResultFile.gdr
                                          with empty report
                                          and empty Result map
```



ADDGRAPHTOPLOT(PLOTNUMBER, X VALUES, Y VALUES, GRAPH TITLE)

Adds a graph to one of the created plots. Enter the X- and Y- values and the graph title. Example:

```
import gdr                                             # import gdr library

gdrf =                                                 # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")     the name
                                                           CustomResultFile.gdr
                                                           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, YLabel, '%', 'µm', XValues,  # add plot to result file
    YValues, GraphName)                                    with plotname,
                                                           labels, units,
                                                           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, YLabel2, '%', '',     # add plot to result file
    XValues2, YValues2, GraphName2)                        with name, labels,
                                                           units, values and
                                                           graph name
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, YValues3, GraphName3)  # add graph to second
                                                           plot

gdrf.saveResults({},"", "2024")                        # write and save the file
                                                           CustomResultFile.gdr
                                                           with empty report
                                                           and empty result map
```



POSTMAP = PYTHON DICTIONARY

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() 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.

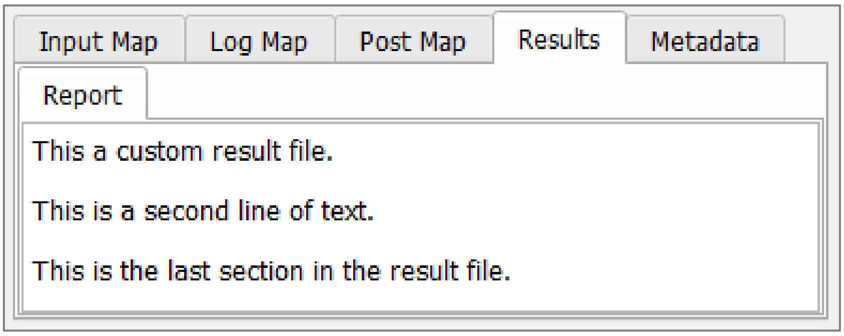
```
import gdr                                            # import gdr library

gdrf =                                                # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")    the name
                                                          CustomResultFile.gdr

plotParameters = {                                    # define plot parameters
   'PlotTitle'  : 'Another Plot',                         and assign them to
   'XAxisLabel' : 'X-Axis Label',                         variable
   'YAxisLabel' : 'Y-Axis Label',                         plotParameters
   'XAxisUnit'  : '',                                 # define labels for the
   'YAxisUnit'  : '%',                                    two axes
   'XAxisRange'      : 'Auto',
   'YAxisRange'      : 'MinMax',                      # define units for both
   'YAxisMinValue'   : -5,                                axes
   'YAxisMaxValue'   : 15,
   'NumberOfGraphs'  : 1,                             # define how the default
   'Graph1' : {                                           axis range should be
     'GraphTitle' : 'This is a plot',                     given. Possible
     'DrawStyle'      : 'Lines',                          values are
     'XValues' : [0,1,2,3],                               Automatic, Auto,
     'YValues' : [0,2,4,9]}}                              MinMax, Tight
                                                      # define number of graphs
                                                          in the plot

                                                      # possible values for
                                                          DrawStyle are
                                                          LinesPoints, Bars,
                                                          Lines, Points,
                                                          FilledStep,
                                                          VerticalSpan,
                                                          HorizontalSpan

postParameters = {                                    # assign Python
   'Plots' : {                                            dictionary
     'NumberOfPlots' : 1,                                 containing plot
     'Plot1'         : plotParameters}}                   parameters to
                                                          variable
                                                          postParameters
gdrf.postMap = postParameters                         # add a Post Map tab and
                                                          plots to result file

gdrf.saveResults({},"", "2024")                       # write and save the file
                                                          CustomResultFile.gdr
                                                          with empty report
                                                          and empty Result map
```

ADDTEXT (STRING)

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. Example:

```
import gdr                                            # import gdr library
```

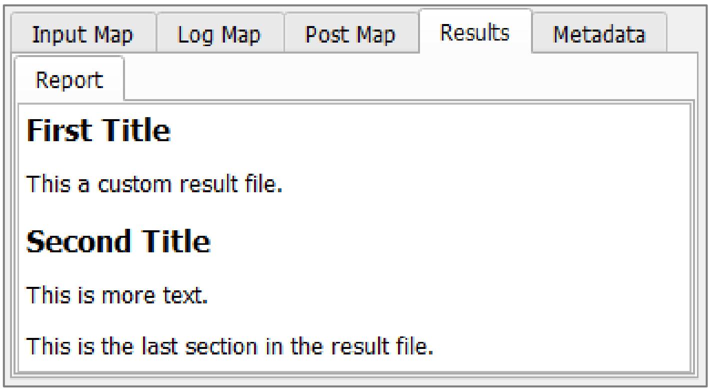
```
gdrf = gdr.GDR.createEmptyResults("CustomResultFile","2024")  # open a result file with the name CustomResultFile.gdr

gdrf.addText("This a custom result file.")                    # add text in Report tab
gdrf.addText("This is a second line of 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, "2024")                          # write and save the file CustomResultFile.gdr with report and an empty Result map
```



ADDTITLE (STRING)

Adds titles in the **Result – Report** subtab of the generated result file. This is a similar to the addText command, but with bold font. Example:

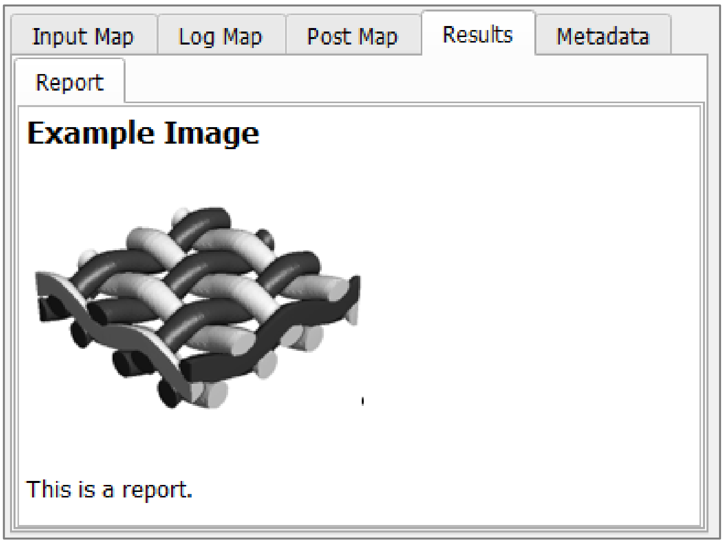
```
import gdr                                                    # import gdr library

gdrf = gdr.GDR.createEmptyResults("CustomResultFile","2024")  # 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, "2024")                          # write and save the file CustomResultFile.gdr with report and an empty Result map
gd.showGDR('CustomResultFile.gdr')                            # open result file in Result Viewer
```



ADDIMAGE(STRING IMAGE FILE PATH, STRING TITLE)

Adds an image to the **Results – Report** subtab of the generated result file. Additionally, the image is saved to the corresponding result folder. Example:

```
import gdr                                                  # import gdr library

gdrf =                                                      # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")          the name
                                                                CustomResultFile.gdr

gdrf.addImage('../image.png', 'Example Image')              # add image to Report tab

report = "This is a report."                                # define text for the
                                                                Report tab and
                                                                assign it to
                                                                variable report
gdrf.saveResults({}, report, "2024")                        # write and save the file
                                                                CustomResultFile.gdr
                                                                with report and an
                                                                empty Result map
```

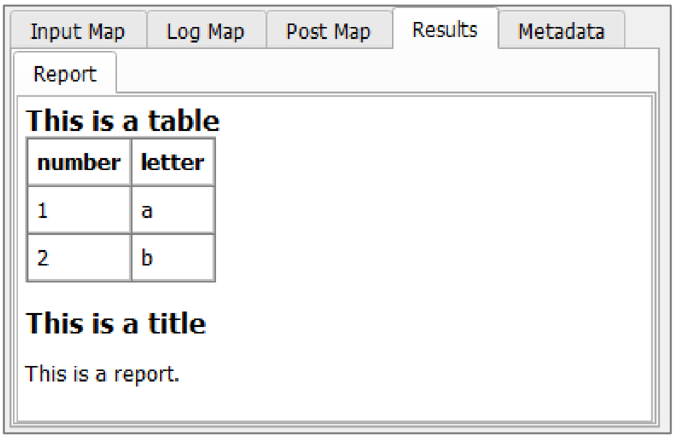


ADDTABLE(STRING TITLE, LIST COLUMN HEADERS, *LIST TABLE)

Adds table to the **Results – Report** subtab of the generated result file. Example:

```
import gdr                                                  # import gdr library

gdrf =                                                      # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")          the name
                                                                CustomResultFile.gdr

col_headers = ["number", "letter"]                          # define column headers
                                                                in a list of strings
```

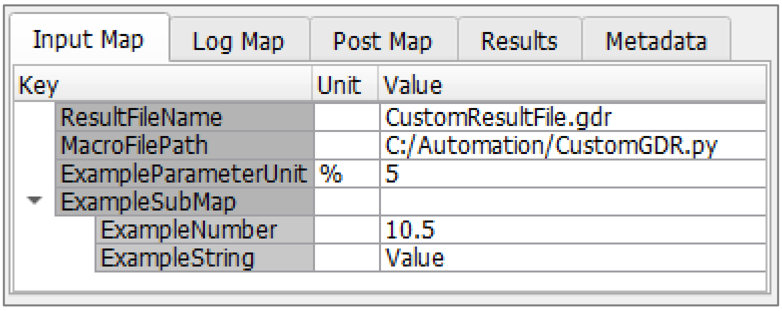
```
table = [[1, 2], ["a", "b"]]                                # define table as list of columns

gdrf.addTable("This is a table", col_headers, *table)       # add table to 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, "2024")                        # write and save the file CustomResultFile.gdr with report and an empty Result map
```



INPUTMAP.UPDATE(PYTHON DICTIONARY)

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. Example:

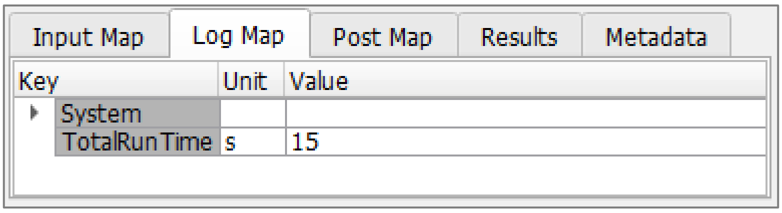
```
import gdr                                                  # import gdr library

gdrf =
 gdr.GDR.createEmptyResults("CustomResultFile","2024")      # open a result file with the name CustomResultFile.gdr

InputParameters = {                                         # assign a Python dictionary containing the input parameters as key-value pairs to variable InputParameters
  'ExampleParameterUnit' :(5,'%'),
  'ExampleSubMap' : {
    'ExampleNumber' : 10.5,
    'ExampleString' : 'Value'}}

gdrf.inputMap.update(InputParameters)                       # add entries to InputMap tab of result file

gdrf.saveResults({},"", "2024")                             # write and save the file CustomResultFile.gdr with empty report and empty Result map
```



LOGMAP.UPDATE(PYTHON DICTIONARY)

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. Example:

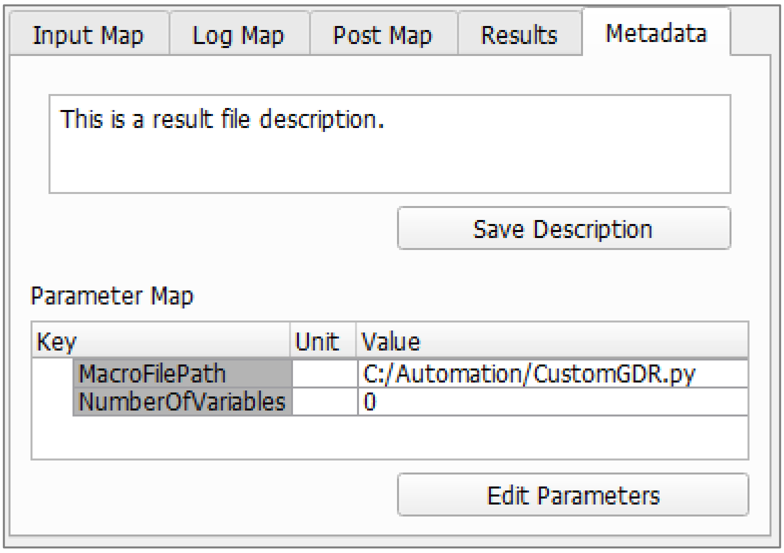
```
import gdr                                              # import gdr library

gdrf =                                                  # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")     the name
                                                           CustomResultFile.gdr

LogParameters = {                                       # assign a Python
  'TotalRunTime' :(15,'s')}                                dictionary
                                                           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({},"", "2024")                         # write and save the file
                                                           CustomResultFile.gdr
                                                           with empty report
                                                           and empty Result map
```



SETDESCRIPTION(STRING DESCRIPTION)

Replaces the default description "Created by macro 'macro file path'." by a custom description in the **Metadata** tab of the generated result file. Example:

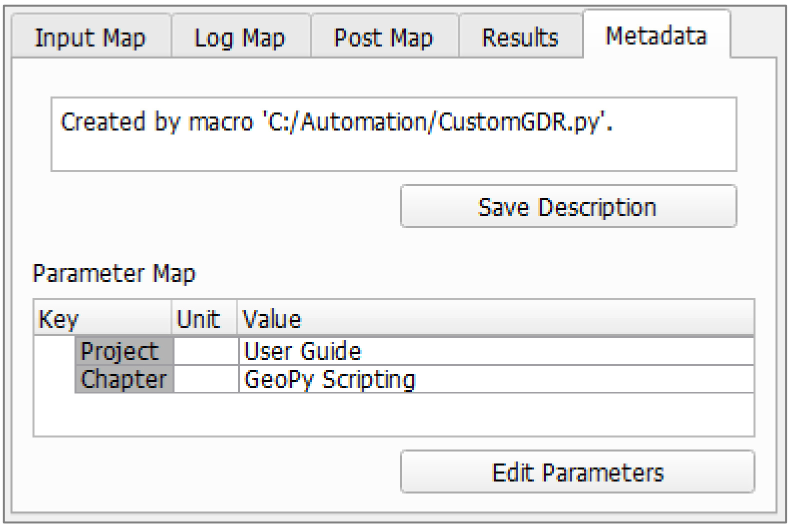
```
import gdr                                              # import gdr library

gdrf =                                                  # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")     the name
                                                           CustomResultFile.gdr

gdrf.setDescription("This is a result file description.") # add description to
                                                           Metadata tab
gdrf.saveResults({},"", "2024")                         # write and save the file
                                                           CustomResultFile.gdr
                                                           with empty report
                                                           and empty Result map
```



PARAMETERMAP.UPDATE(PYTHON DICTIONARY)

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. Example:

```
import gdr                                            # import gdr library

gdrf =                                                # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")    the name
                                                          CustomResultFile.gdr

ParameterParameters = {                               # assign a Python
  'Project' : 'User Guide',                               dictionary
  'Chapter' : 'GeoPy Scripting'}                          containing
                                                          parameters to the
                                                          variable
                                                          ParameterParameters

gdrf.parameterMap = ParameterParameters               # add a Parameter Map to
                                                          result file

gdrf.saveResults({},"", "2024")                       # write and save the file
                                                          CustomResultFile.gdr
                                                          with empty report
                                                          and empty Result map
```



COMMAND = STRING 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. Example:

```
import gdr                                              # import gdr library

gdrf =                                                  # open a result file with
 gdr.GDR.createEmptyResults("CustomResultFile","2024")      the name
                                                            CustomResultFile.gdr

gdrf.command = 'ExampleModule:ExampleSolver'            # add command with module
                                                            name ExampleModule
                                                            and solver name
                                                            ExampleSolver to
                                                            result file

gdrf.saveResults({},"", "2024")                         # write and save the file
                                                            CustomResultFile.gdr
                                                            with empty report
                                                            and empty Result map
```

	File	Module	Command
1	C:/Automation/CustomResultFile.gdr	ExampleModule	ExampleSolver

Access to GeoDict structures and result fields (GUF 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.

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.

The **Creation Data** block provides information about the creation of the file, e.g. the creation time and the used GeoDict revision.

```
GufVersion GDT3
HeaderLength 4608
DateOfCreation Fri Nov 17 2023 15:56:22
GeoDictEdition 2024
GeoDictRevision 69788
GeoDictPlatform 64 bit Windows
```

Creation 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
```

Image Data

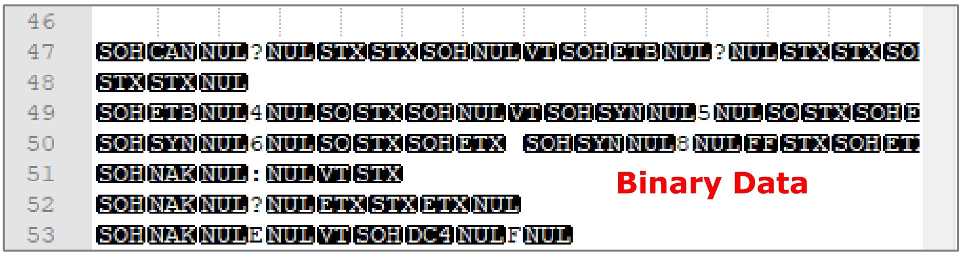
At the end of the header, **File Specific Data** blocks can be found, e.g., map data, info data, and array data.

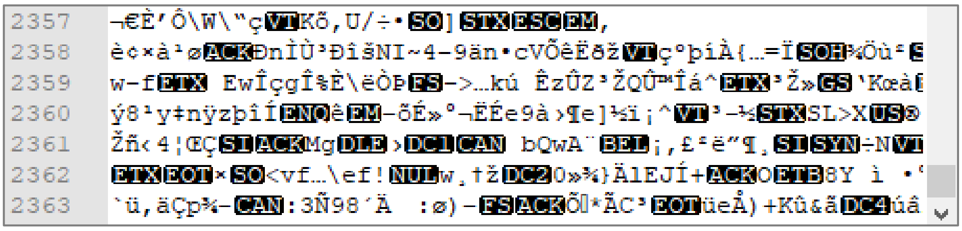
```
NumberOfMaps 4
NamesOfMaps GAD,GADStats,Materials,MaterialDatabase
Map1:Compression zlib
Map1:Offset 63650
Map1:Length 4224
Map2:Compression zlib
Map2:Offset 67874
Map2:Length 231
Map3:Compression zlib
Map3:Offset 68105
Map3:Length 1020
Map4:Compression zlib
Map4:Offset 69125
Map4:Length 2102
```

File Specific 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.

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.





In a second example, a flow simulation was run on the structure. The GUF file FlowField_z.vap file is produced and shown here.

```
GufVersion VAP1
HeaderLength 1024
DateOfCreation 17.11.23  15:56:21
GeoDictEdition 2024
GeoDictRevision 69788
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
```

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.

Result files generated by the particle tracker in FilterDict and AddiDict (*.gpp) contain a large information block providing details about the simulation.

```
GufVersion GPP3
HeaderLength 13312
DateOfCreation Fri Nov 17 2023 16:02:14
GeoDictEdition 2024
GeoDictRevision 69788
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 3600
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 277200
Info:TotalMultiplicitySum 3600
```

The particle positions are described by arrays. The example file contains one array with 12 columns and 3600 rows.

Access GUF files with GeoPy

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.

Example:

```
from guf import GUF                                  # import GUF library
guf_file = GUF("StokesResult/FlowField_z.vap")       # access GUF file 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 = guf_header["NumberOfImages"]           # assign the number of images to the
                                                         variable imagenumber
gd.msgBox(f"The file contains {imagenumber}          # show message dialog
    images.")
```

getImage(string name)

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.

```
NameOfCreator EJStokes
NamesOfImages Velocity,Pressure
NumberOfImages 2
```

```
Image1:Meaning vector
Image1:Names VelocityX,VelocityY,VelocityZ
Image1:Types float,float,float
```

Basically, this function does the same as the gd.getVolumeField() function described above, but here no volume field needs to be loaded in GeoDict.

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** or the **gd.getVolumeField** functions, described on pages 58 and 60 respectively.

```
Image1:Units 1
Image1:Compression rle
Image1:Offset 4608
```

Example:

```
from guf import GUF                                  # import GUF library

guf_file = GUF("StokesResult/FlowField_z.vap")       # access GUF file FlowField_z.vap

guf_image = guf_file.getImage("Velocity")            # assign the numpy array
                                                       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 position (50,50,50)      # show a message dialog for the
    in the Velocity X field is                         velocity at position (50,50,50)
    {guf_field[50][50][50]}.")
```

GETARRAY(STRING NAME)

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.

```
NumberOfArrays 1
NamesOfArrays ParticlePositions
Array1:NumberOfColumns 12
Array1:NumberOfRows 3600
Array1:ColumnNames ID,Type,Position X,Position Y,Position Z,Velocity
X,Velocity Y,Velocity Z,Time,Collision Count,Status,Multiplicity
```

This function only works, if the GUF file contains arrays (e.g. FilterDict *.gpp files). There are many very helpful **FilterDict Particle specific Functions** described on pages 72ff, but for the getArray function the trajectories do not need to be loaded in GeoDict.

Example:

```
from guf import GUF                                       # import GUF library

guf_file =                                                # access FilterDict result
    GUF("FilterLifeTime/Batch00001/TrackerFinalParticles    file
    .gpp")                                                  TrackerFinalParticles.
                                                            gpp

guf_array = guf_file.getArray("ParticlePositions")        # assign the numpy array
                                                            containing the
                                                            particle 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 {id_5} has the X-position  # show message dialog
    {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 {guf_row[0]} has the X-position {guf_row[2]} at time {guf_row[8]}.")   # show the same message dialog as before
```

GETMAP(STRING NAME)

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.

```
NumberOfMaps 4
NamesOfMaps GAD,GADStats,Materials,MaterialDatabase
Map1:Compression zlib
```

There are many very helpful **General Functions** described on pages 50ff applicable for structure files (e.g. gd.getGADObject), 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.

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.

Example:

```
from guf import GUF                               # import GUF library

guf_file = GUF("FiberGeo/Structure.gdt")          # access GUF file Structure.gdt in the folder FiberGeo

guf_map = guf_file.getMap("GADStats")             # assign the stringmap of the GAD statistics for all 2D slices to the variable guf_map

print(guf_map)                                    # print the stringmap to the GeoDict console

objectscount_Z =
    guf_map["PerSliceObjectCountsZ"]              # assign the string containing statistics for the Z-slices to the variable objectscount_Z

objectscount_Z = objectscount_Z.split(',')        # split the string by commas, and obtain a list

count_Z_slice_13 = objectscount_Z[13]             # assign the 14th entry in the list (index 13 as counting starts with 0) to the variable count_Z_slice

gd.msgBox(f" In Z-slice 14 there are
    {count_Z_slice_13} objects.")                 # show a message dialog.
```

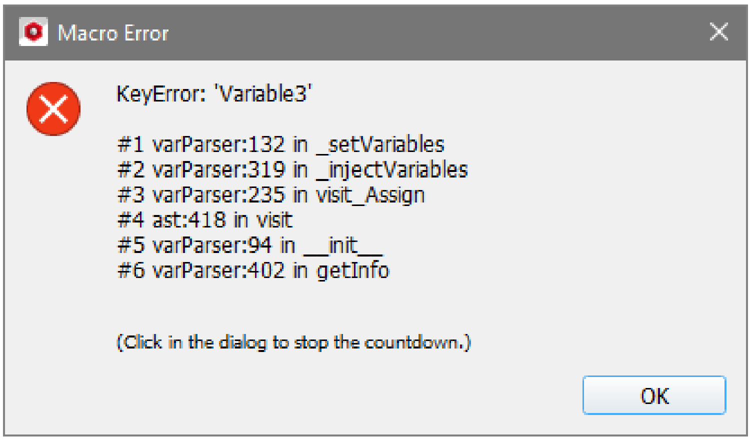
Error reporting

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.

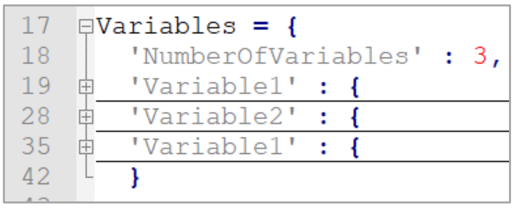
Variables Dictionary

KeyError: 'Variable#'

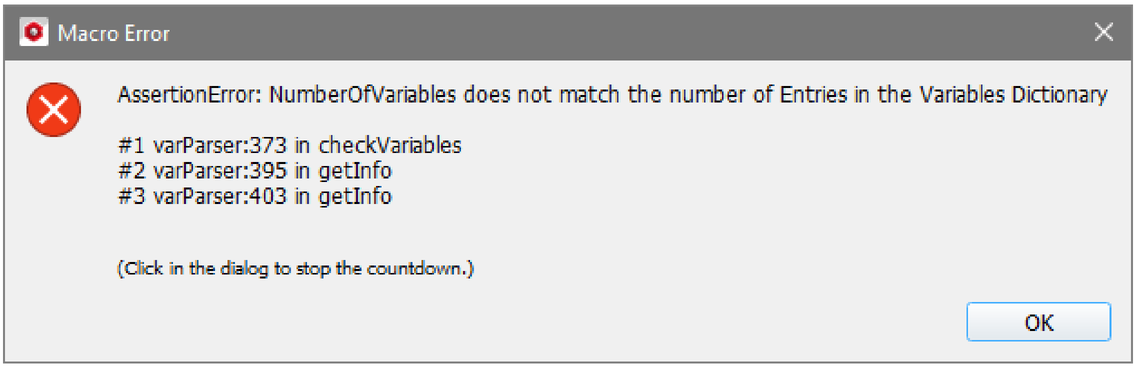


There are not as many variables given in the dictionary as given by the NumberOfVariables entry in the Variables dictionary.

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

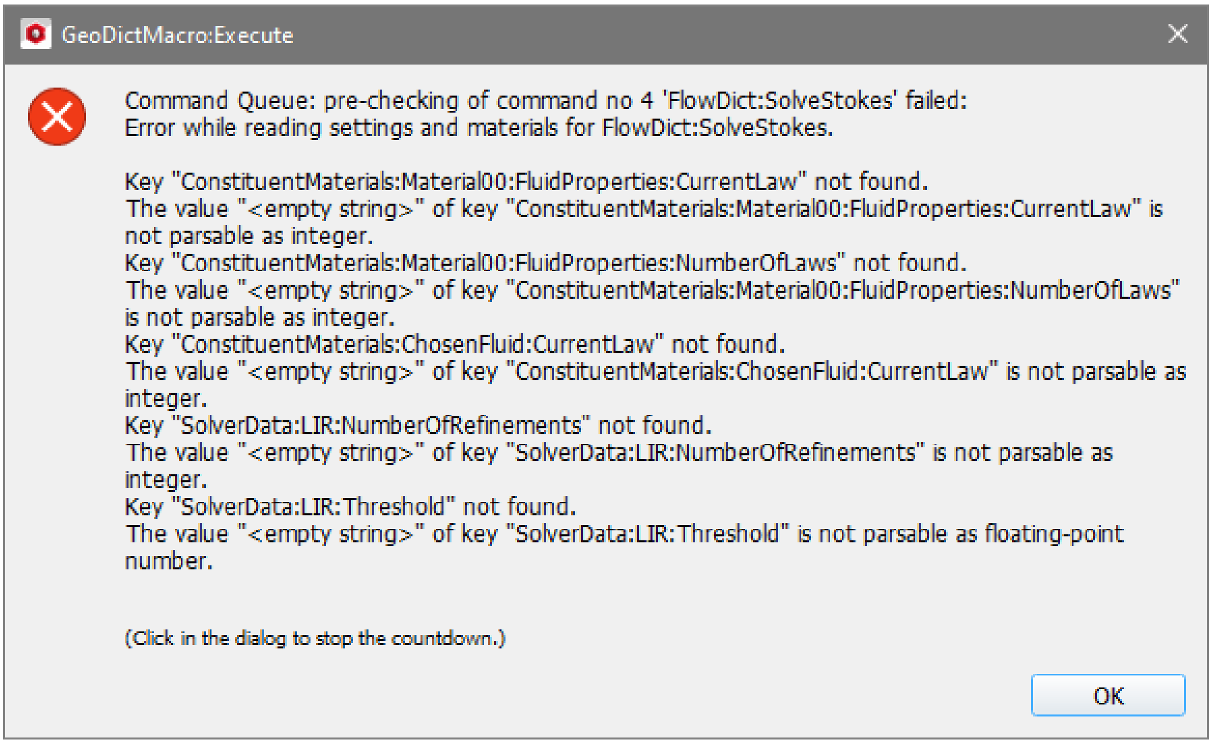


This error happens, if there are more variable entries, than the NumberOfVariables parameter determines.

```
Variables = {
    'NumberOfVariables' : 2,
    'Variable1' : {
    'Variable2' : {
    'Variable3' : {
    }
```

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.

```
Header =  {
    'Release'        : '2024',
    }
```

```
gd.runCmd("FlowDict:SolveStokes", SolveStokes_args_1, Header['Release'])
```

```
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'])
```

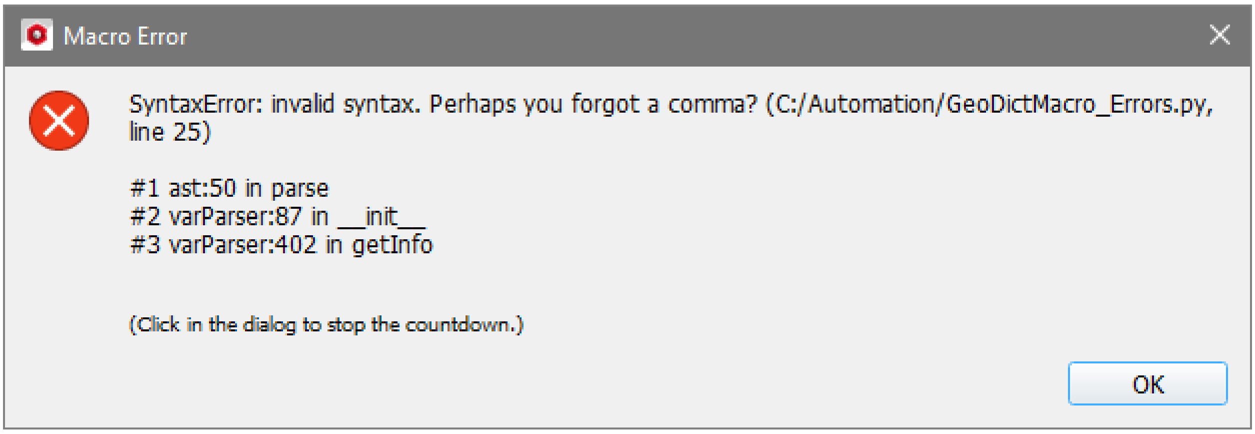
```
gd.runCmd("FiberGeo:Create", Create_args_1, Header['Release'])
```

Invalid Syntax

SyntaxError: … (macrofilepath, line)

There are many possibilities to obtain a syntax error. Some of the most common syntax errors are:

- **Perhaps you forgot a comma?**: missing commas in the end of a line in a Python dictionary




```
17 Variables = {
18   'NumberOfVariables' : 3,
19   'Variable1' : {
20     'Name'           : 'gd_SVP',
21     'Label'          : 'Solid Volume Percentage',
22     'Type'           : 'double',
23     'Unit'           : '%',
24     'ToolTip'        : 'Solid volume percentage of the created structure.',
25     'BuiltinDefault' : 10.0
26     'Check'          : 'min0;max100'
27   },
```

- **'{' was never closed**: missing closing brackets

Macro Error

SyntaxError: '{' was never closed (C:/Automation/GeoDictMacro_Errors.py, line 17)

#1 ast:50 in parse
#2 varParser:87 in __init__
#3 varParser:402 in getInfo

(Click in the dialog to stop the countdown.)

OK

```
17 Variables = {
18   'NumberOfVariables' : 3,
19   'Variable1' : {
20     'Name'           : 'gd_SVP',
21     'Label'          : 'Solid Volume Percentage',
22     'Type'           : 'double',
23     'Unit'           : '%',
24     'ToolTip'        : 'Solid volume percentage of the created structure.',
25     'BuiltinDefault' : 10.0,
26     'Check'          : 'min0;max100',
27     
28   'Variable2' : {
```

- **unterminated string literal**: missing quotation ending

Macro Error

SyntaxError: unterminated string literal (detected at line 26) (C:/Automation/GeoDictMacro_Errors.py, line 26)

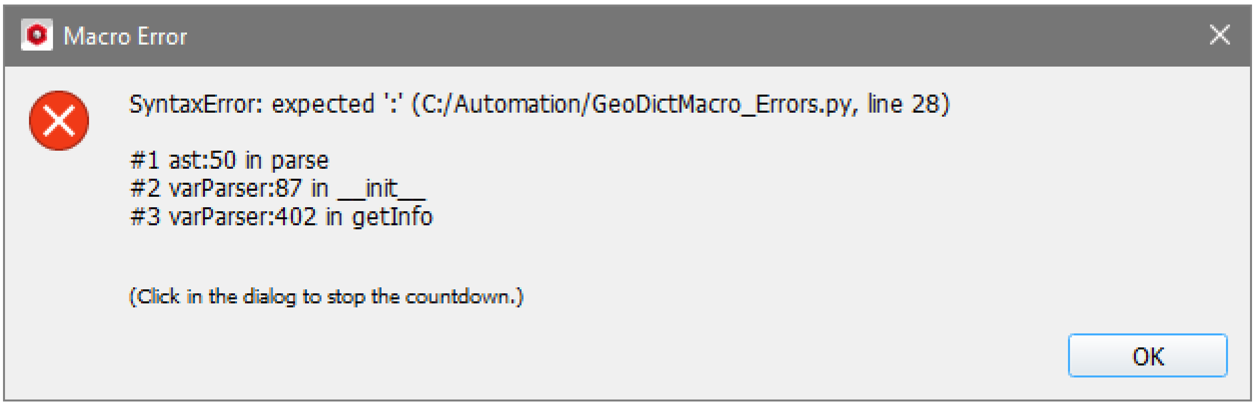
#1 ast:50 in parse
#2 varParser:87 in __init__
#3 varParser:402 in getInfo

(Click in the dialog to stop the countdown.)

OK

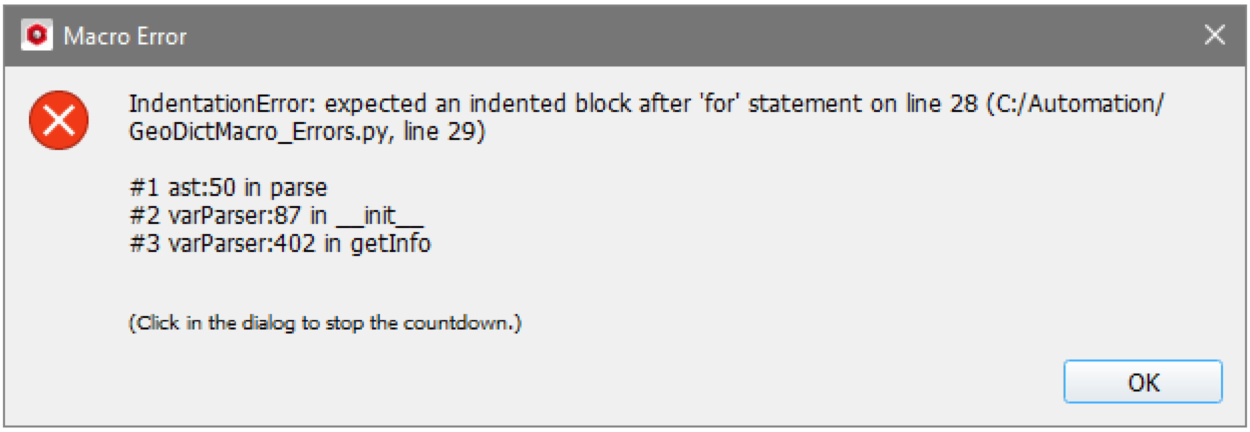
```
17 Variables = {
18   'NumberOfVariables' : 3,
19   'Variable1' : {
20     'Name'           : 'gd_SVP',
21     'Label'          : 'Solid Volume Percentage',
22     'Type'           : 'double',
23     'Unit'           : '%',
24     'ToolTip'        : 'Solid volume percentage of the created structure.',
25     'BuiltinDefault' : 10.0,
26     'Check'          : 'min0;max100,
27     }
```

- **expected ':'**:missing colons (e.g. in definitions or in the line defining a loop.




```
for i in range(5)
    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).

```
for i in range(5):
print(i)
```

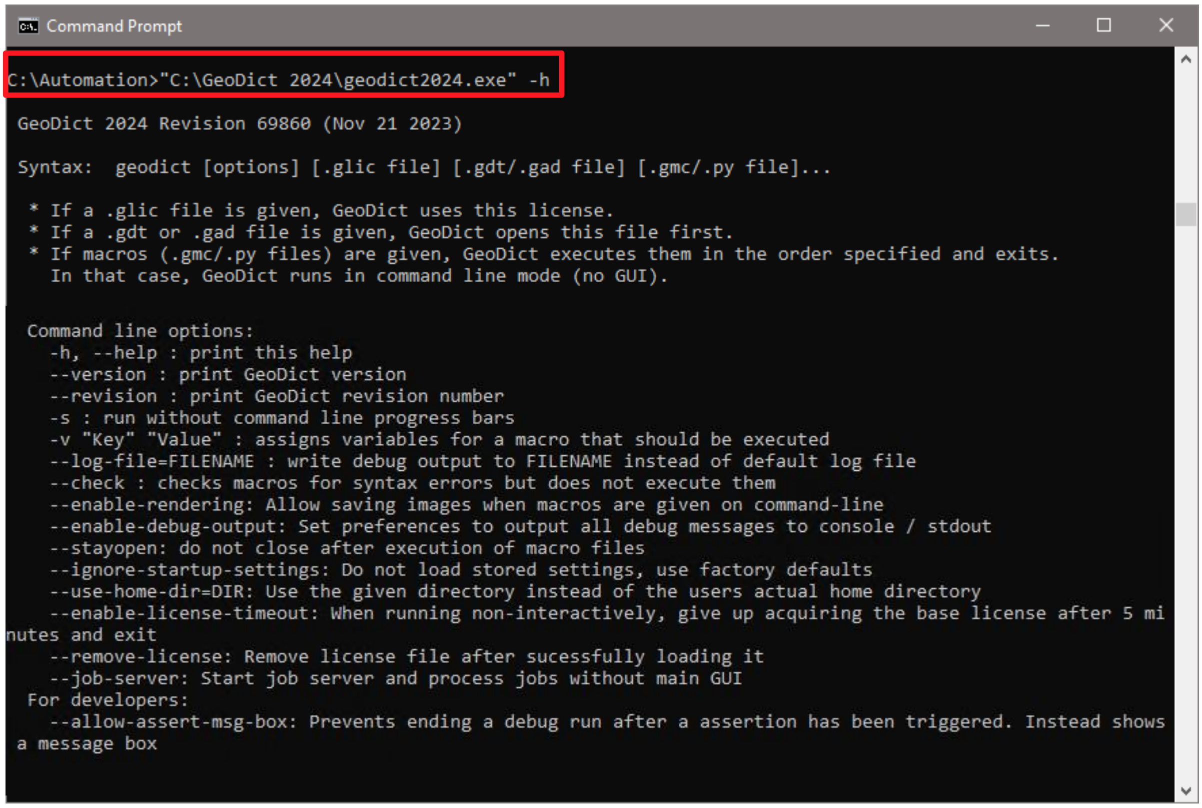
Execute a Python script

Python scripts are executed as shown above starting in page 8 (script without variables) and starting in page 14 (script with variables) for GeoPy macros.

Running GeoDict from the Command Line

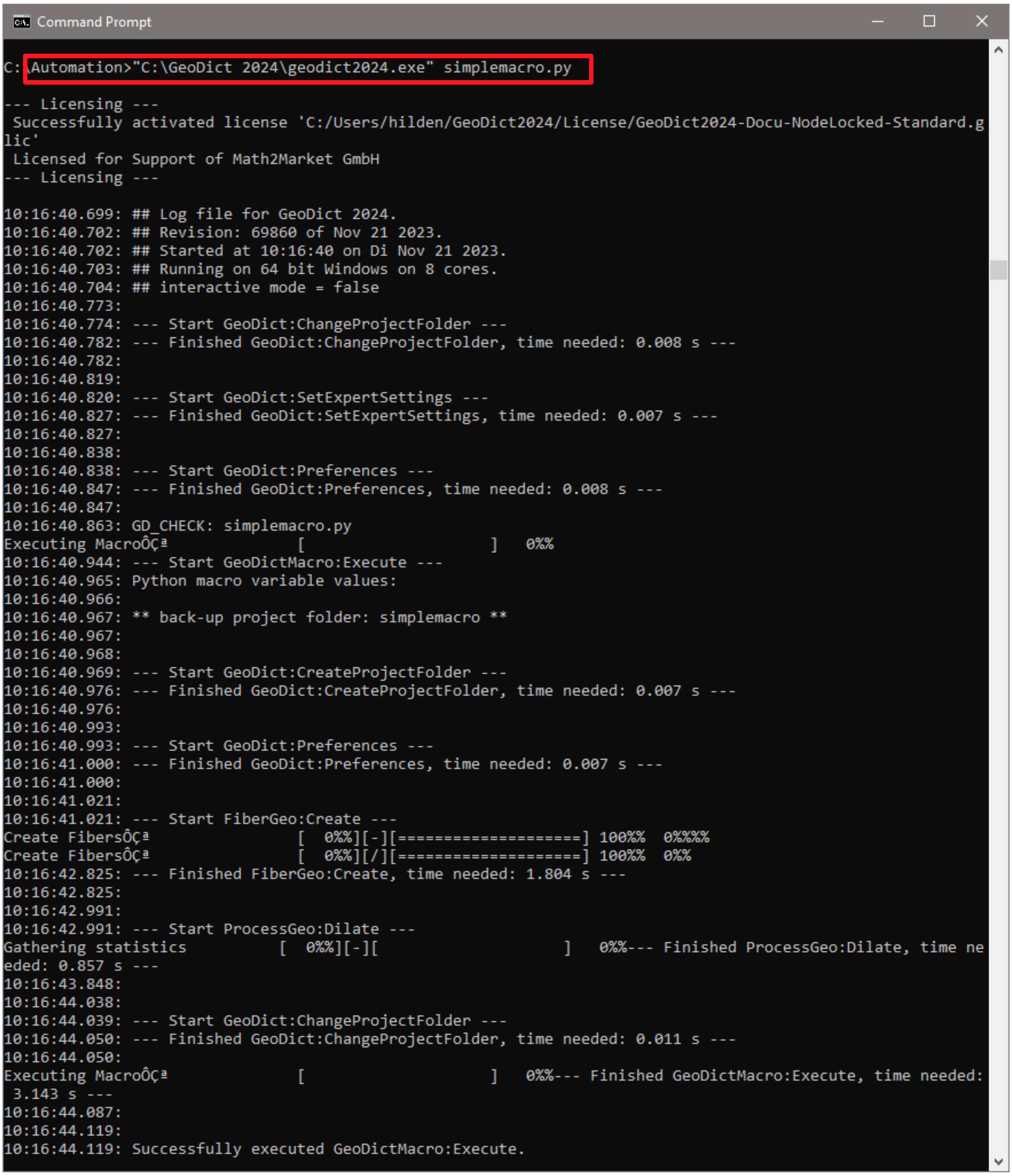
Being comfortable with the command prompt, it is a fast possibility to run GeoDict from the command line without the GUI. Although it is possible to open GeoDict from the command line (**>>Installationpath\geodict2024.exe**), it is not necessary for running macros. The following command prints a helpful list of commands:

>>"Installation-path\geodict2024.exe" -h



Macros can be executed using the command

>>"Installation-path\geodict2024.exe" macro-file



```
C:\Automation>"C:\GeoDict 2024\geodict2024.exe" simplemacro.py

--- Licensing ---
 Successfully activated license 'C:/Users/hilden/GeoDict2024/License/GeoDict2024-Docu-NodeLocked-Standard.g
lic'
 Licensed for Support of Math2Market GmbH
--- Licensing ---

10:16:40.699: ## Log file for GeoDict 2024.
10:16:40.702: ## Revision: 69860 of Nov 21 2023.
10:16:40.702: ## Started at 10:16:40 on Di Nov 21 2023.
10:16:40.703: ## Running on 64 bit Windows on 8 cores.
10:16:40.704: ## interactive mode = false
10:16:40.773:
10:16:40.774: --- Start GeoDict:ChangeProjectFolder ---
10:16:40.782: --- Finished GeoDict:ChangeProjectFolder, time needed: 0.008 s ---
10:16:40.782:
10:16:40.819:
10:16:40.820: --- Start GeoDict:SetExpertSettings ---
10:16:40.827: --- Finished GeoDict:SetExpertSettings, time needed: 0.007 s ---
10:16:40.827:
10:16:40.838:
10:16:40.838: --- Start GeoDict:Preferences ---
10:16:40.847: --- Finished GeoDict:Preferences, time needed: 0.008 s ---
10:16:40.847:
10:16:40.863: GD_CHECK: simplemacro.py
Executing MacroÔÇª               [                    ]   0%%
10:16:40.944: --- Start GeoDictMacro:Execute ---
10:16:40.965: Python macro variable values:
10:16:40.966:
10:16:40.967: ** back-up project folder: simplemacro **
10:16:40.967:
10:16:40.968:
10:16:40.969: --- Start GeoDict:CreateProjectFolder ---
10:16:40.976: --- Finished GeoDict:CreateProjectFolder, time needed: 0.007 s ---
10:16:40.976:
10:16:40.993:
10:16:40.993: --- Start GeoDict:Preferences ---
10:16:41.000: --- Finished GeoDict:Preferences, time needed: 0.007 s ---
10:16:41.000:
10:16:41.021:
10:16:41.021: --- Start FiberGeo:Create ---
Create FibersÔÇª                 [  0%%][-][====================] 100%%  0%%%%
Create FibersÔÇª                 [  0%%][/][====================] 100%%  0%%
10:16:42.825: --- Finished FiberGeo:Create, time needed: 1.804 s ---
10:16:42.825:
10:16:42.991:
10:16:42.991: --- Start ProcessGeo:Dilate ---
Gathering statistics          [  0%%][-][                    ]   0%%--- Finished ProcessGeo:Dilate, time ne
eded: 0.857 s ---
10:16:43.848:
10:16:44.038:
10:16:44.039: --- Start GeoDict:ChangeProjectFolder ---
10:16:44.050: --- Finished GeoDict:ChangeProjectFolder, time needed: 0.011 s ---
10:16:44.050:
Executing MacroÔÇª               [                    ]   0%%--- Finished GeoDictMacro:Execute, time needed:
 3.143 s ---
10:16:44.087:
10:16:44.119:
10:16:44.119: Successfully executed GeoDictMacro:Execute.
```

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 parameter macro use **-v "Key" "Value"** for each variable.

```
Command Prompt
C:\Automation>"C:\GeoDict 2024\geodict2024.exe" VariableStudy.py -v "gd_SVP" "5" -v "gd_RandomSeed" "15"

--- Licensing ---
 Successfully activated license 'C:/Users/hilden/GeoDict2024/License/GeoDict2024-Docu-NodeLocked-Standard.g
lic'
 Licensed for Support of Math2Market GmbH
--- Licensing ---

10:14:15.327: ## Log file for GeoDict 2024.
10:14:15.327: ## Revision: 69860 of Nov 21 2023.
10:14:15.328: ## Started at 10:14:15 on Di Nov 21 2023.
10:14:15.328: ## Running on 64 bit Windows on 8 cores.
10:14:15.329: ## interactive mode = false
10:14:15.387:
10:14:15.388: --- Start GeoDict:ChangeProjectFolder ---
10:14:15.395: --- Finished GeoDict:ChangeProjectFolder, time needed: 0.007 s ---
10:14:15.396:
10:14:15.432:
10:14:15.432: --- Start GeoDict:SetExpertSettings ---
10:14:15.439: --- Finished GeoDict:SetExpertSettings, time needed: 0.007 s ---
10:14:15.439:
10:14:15.451:
10:14:15.451: --- Start GeoDict:Preferences ---
10:14:15.458: --- Finished GeoDict:Preferences, time needed: 0.006 s ---
10:14:15.458:
10:14:15.470: GD_CHECK: VariableStudy.py
Executing MacroÔÇª              [                    ]   0%%
10:14:15.547: --- Start GeoDictMacro:Execute ---
10:14:15.565: found value: 5.0
10:14:15.566: found value: 15
10:14:15.570: found value: 10.0
10:14:15.572: Python macro variable values:
10:14:15.572: gd_SVP  =  [5.0, '%']
10:14:15.573: gd_RandomSeed  =  [15, '']
10:14:15.573: gd_FiberDiameter  =  [10.0, '┬Ám']
10:14:15.575:
10:14:15.575: --- Start GeoDict:ChangeProjectFolder ---
10:14:15.584: --- Finished GeoDict:ChangeProjectFolder, time needed: 0.008 s ---
10:14:15.584:
10:14:15.596:
10:14:15.596: ** back-up GeoDict results: FiberGeo_5.0_15_10.0.gdr **
10:14:15.598:
10:14:15.606:
10:14:15.606: --- Start FiberGeo:Create ---
Create FibersÔÇª                [  0%%][/][====================] 100%%  0%%%%
Create FibersÔÇª                [  0%%][\][====================] 100%%  0%%
10:14:16.878: --- Finished FiberGeo:Create, time needed: 1.271 s ---
10:14:16.878:
10:14:17.023:
10:14:17.023: --- Start ProcessGeo:Dilate ---
Gathering statistics          [  0%%][/][                    ]   0%%--- Finished ProcessGeo:Dilate, time ne
eded: 0.871 s ---
10:14:17.894:
10:14:18.043:
10:14:18.043: --- Start GeoDict:ChangeProjectFolder ---
10:14:18.056: --- Finished GeoDict:ChangeProjectFolder, time needed: 0.012 s ---
10:14:18.056:
Executing MacroÔÇª              [                    ]   0%%--- Finished GeoDictMacro:Execute, time needed:
 2.546 s ---
10:14:18.093:
10:14:18.124:
```

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.

```
Command Prompt
C:\Automation>"C:\GeoDict 2024\geodict2024.exe" saveImage.py --enable-rendering
```

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

Technical documentation:

Janine Hilden

Jürgen Becker

Anne Blumer

Barbara Planas

MATH 2 MARKET

Math2Market GmbH

Richard-Wagner-Str. 1, 67655 Kaiserslautern, Germany
www.geodict.com