ogs5py Documentation

Release 1.1.1

Sebastian Mueller

Apr 02, 2020
# Contents

## 1 ogs5py Quickstart

1.1 Installation ........................................... 1
1.2 Further Information ................................... 1
1.3 Pumping Test Example ................................ 1
1.4 OGS executable ...................................... 3
1.5 Requirements ......................................... 3
1.6 License .................................................. 4

## 2 ogs5py Tutorials

2.1 Tutorial 1: A pumping test ............................ 5
2.2 Tutorial 2: Interaction with GSTools ................. 7
2.3 Tutorial 2: Interaction with pygmsh ................. 10

## 3 ogs5py API

3.1 Purpose .................................................. 13
3.2 Subpackages ............................................. 13
3.3 Classes .................................................. 13
   OGS model Base Class ................................ 13
   File Classes ........................................... 13
3.4 Functions .............................................. 14
   Geometric ............................................. 14
   Searching ............................................ 14
   Formatting ........................................... 14
   Downloading ......................................... 15
   Plotting .............................................. 15
   Information ........................................... 15
3.5 ogs5py.ogs .............................................. 16
   OGS Class ............................................ 16
3.6 ogs5py.fileclasses .................................... 25
   Subpackages .......................................... 25
   File Classes ......................................... 25
   ogs5py.fileclasses.base .............................. 26
   ogs5py.fileclasses.asc ............................... 36
   ogs5py.fileclasses.bc ............................... 39
   ogs5py.fileclasses.cct .............................. 46
   ogs5py.fileclasses.dde .............................. 52
   ogs5py.fileclasses.fct .............................. 58
   ogs5py.fileclasses.gem .............................. 64
   ogs5py.fileclasses.gli .............................. 72
   ogs5py.fileclasses.ic ............................... 84
   ogs5py.fileclasses.krc .............................. 93
   ogs5py.fileclasses.mcp .............................. 101
ogs5py is a Python-API for the OpenGeoSys 5 scientific modeling package.

1.1 Installation

The package can be installed via pip. On Windows you can install WinPython to get Python and pip running.

```
pip install ogs5py
```

1.2 Further Information

- General homepage: https://www.opengeosys.org/ogs-5
- OGS5 Repository: https://github.com/ufz/ogs5
- Keyword documentation: https://ogs5-keywords.netlify.com
- OGS5 Benchmarks: https://github.com/ufz/ogs5-benchmarks
- ogs5py Benchmarks: https://github.com/GeoStat-Framework/ogs5py_benchmarks

1.3 Pumping Test Example

In the following a simple transient pumping test is simulated on a radial symmetric mesh. The point output at the observation well is plotted afterwards.
from ogs5py import OGS, specialrange, generate_time
from matplotlib import pyplot as plt

# discretization and parameters
time = specialrange(0, 3600, 50, typ="cub")
radii = specialrange(0, 1000, 100, typ="cub")
obs = radii[21]
angles = 32
storage = 1e-3
transmissivity = 1e-4
rate = -1e-3

# model setup
model = OGS(task_root="pump_test", task_id="model")

# generate a radial mesh and geometry ("boundary" polyline)
model.msh.generate("radial", dim=2, radii, angles=angles)
model.gli.generate("radial", dim=2, radii[-1], angles=angles)
model.gli.add_points([0.0, 0.0, 0.0], "pwell")
model.gli.add_points([obs, 0.0, 0.0], "owell")
model.bc.add_block(
    # boundary condition
    PCS_TYPE="GROUNDWATER_FLOW",
    PRIMARY_VARIABLE="HEAD",
    GEO_TYPE=["POLYLINE", "boundary"],
    DIS_TYPE=["CONSTANT", 0.0],
)
model.st.add_block(
    # source term
    PCS_TYPE="GROUNDWATER_FLOW",
    PRIMARY_VARIABLE="HEAD",
    GEO_TYPE=["POINT", "pwell"],
    DIS_TYPE=["CONSTANT_NEUMANN", rate],
)
model.ic.add_block(
    # initial condition
    PCS_TYPE="GROUNDWATER_FLOW",
    PRIMARY_VARIABLE="HEAD",
    GEO_TYPE="DOMAIN",
    DIS_TYPE=["CONSTANT", 0.0],
)
model.mmp.add_block(
    # medium properties
    GEOMETRY_DIMENSION=2,
    STORAGE=[1, storage],
    PERMEABILITY_TENSOR=["ISOTROPIC", transmissivity],
)
model.num.add_block(
    # numerical solver
    PCS_TYPE="GROUNDWATER_FLOW",
    LINEAR_SOLVER=[2, 5, 1e-14, 1000, 1.0, 100, 4],
)
model.out.add_block(
    # point observation
    PCS_TYPE="GROUNDWATER_FLOW",
    NOD_VALUES="HEAD",
    GEO_TYPE=["POINT", "owell"],
    DAT_TYPE="TECPLOT",
)
model.pcs.add_block(
    # set the process type
    PCS_TYPE="GROUNDWATER_FLOW", NUM_TYPE="NEW"
)
model.tim.add_block(
    # set the timesteps
    PCS_TYPE="GROUNDWATER_FLOW",
    **generate_time(time)
)
model.write_input()
model.run_model()
1.4 OGS5 executable

To obtain an OGS5 executable, ogs5py brings a download routine `download ogs`:

```python
from ogs5py import download_ogs
download_ogs()
```

Then a executable is stored in the ogs5py config path and will be called when a model is run.

You can pass a `version` statement to the `download_ogs` routine, to obtain a specific version (5.7, 5.7.1 (win only) and 5.8). Also “latest” and “stable” are possible. For OGS 5.7 there are executables for Windows/Linux and MacOS. For “5.8”, “latest” and “stable” there are no MacOS pre-builds. Have a look at the documentation for all options.

If you have compiled your own OGS5 version, you can add your executable to the ogs5py config path with `add_exe`:

```python
from ogs5py import add_exe
add_exe("path/to/your/ogs/exe")
```

Otherwise you need to specify the path to the executable within the run command:

```python
model.run_model(ogs_exe="path/to/ogs")
```

1.5 Requirements

- NumPy >= 1.14.5
- Pandas >= 0.23.2
- meshio >= 4.0.3; <5.0
- lxml >= 4.0; <5.0
- pexpect >= 4.0; <5.0
- vtk >= 8.1
1.6 License

MIT
In the following you will find several Tutorials on how to use ogs5py to explore its whole beauty and power.

2.1 Tutorial 1: A pumping test

This is a minimal example on how to setup a pumping test with ogs5py. The result was plotted against the analytical solution.

In this example we use the `generate_time` function, to use an array of time points for the time stepping definition.

```python
model.tim.add_block(**generate_time(time))
```

is equivalent to:

```python
model.tim.add_block(
    TIME_START=0,
    TIME_END=time[-1],
    TIME_STEPS=[
        [1, time[0]],
        [1, time[1]],
        [1, time[2]],
        # ...
    ],
)
```

The script:

```python
import anaflow as ana
from ogs5py import OGS, specialrange, generate_time
from matplotlib import pyplot as plt

# discretization and parameters
time = specialrange(0, 3600, 50, typ="cub")
rad = specialrange(0, 1000, 100, typ="cub")
obs = rad[21]
angles = 32
storage = 1e-3
transmissivity = 1e-4
rate = -1e-3
# model setup
```

(continues on next page)
model = OGS(task_root="pump_test", task_id="model")
model.pcs.add_block(  # set the process type
    PCS_TYPE="GROUNDWATER_FLOW", NUM_TYPE="NEW"
)
# generate a radial mesh and geometry ("boundary" polyline)
model.msh.generate("radial", dim=2, rad=rad, angles=angles)
model.gli.generate("radial", dim=2, rad_out=rad[-1], angles=angles)
model.gli.add_points([0.0, 0.0, 0.0], "pwell")
model.gli.add_points([obs, 0.0, 0.0], "owell")
model.bc.add_block(  # boundary condition
    PCS_TYPE="GROUNDWATER_FLOW",
    PRIMARY_VARIABLE="HEAD",
    GEO_TYPE=["POLYLINE", "boundary"],
    DIS_TYPE=["CONSTANT", 0.0],
)
model.ic.add_block(  # initial condition
    PCS_TYPE="GROUNDWATER_FLOW",
    PRIMARY_VARIABLE="HEAD",
    GEO_TYPE="DOMAIN",
    DIS_TYPE=["CONSTANT", 0.0],
)
model.st.add_block(  # source term
    PCS_TYPE="GROUNDWATER_FLOW",
    PRIMARY_VARIABLE="HEAD",
    GEO_TYPE=["POINT", "pwell"],
    DIS_TYPE=["CONSTANT_NEUMANN", rate],
)
model.mmp.add_block(  # medium properties
    GEOMETRY_DIMENSION=2,
    STORAGE=[1, storage],
    PERMEABILITY_TENSOR=["ISOTROPIC", transmissivity],
)
model.num.add_block(  # numerical solver
    PCS_TYPE="GROUNDWATER_FLOW",
    LINEAR_SOLVER=[2, 5, 1e-14, 1000, 1.0, 100, 4],
)
model.out.add_block(  # point observation
    PCS_TYPE="GROUNDWATER_FLOW",
    NOD_VALUES="HEAD",
    GEO_TYPE=["POINT", "owell"],
    DAT_TYPE="TECPLOT",
)
model.tim.add_block(  # set the timesteps
    PCS_TYPE="GROUNDWATER_FLOW",
    **generate_time(time)  # generate input from time-series
)
model.write_input()
success = model.run_model()
print("success:", success)
# observation
point = model.readtec_point(pcs="GROUNDWATER_FLOW")
time = point["owell"]["TIME"]
head = point["owell"]["HEAD"]
# analytical solution
head_ana = ana.theis(time, obs, storage, transmissivity, rate=rate)
# comparison plot
plt.scatter(time, head, color="k", label="simulated, r={04.2f}m".format(obs))
plt.plot(time, head_ana, label="analytical solution")
plt.xscale("symlog", linthreshx=10, subsx=range(1, 10))
plt.xlim([0, 1.1 * time[-1]])
plt.xlabel("time in s")
2.2 Tutorial 2: Interaction with GSTools

In this example we are generating a log-normal distributed conductivity field on a generated mesh with the aid of GSTools and perform a steady pumping test.

```python
import numpy as np
from ogs5py import OGS, by_id, show_vtk
from gstools import SRF, Gaussian

# covariance model for conductivity field
cov_model = Gaussian(dim=3, var=2, len_scale=10, anis=[1, 0.2])
srf = SRF(model=cov_model, mean=-9, seed=1000)

# model setup
model = OGS(task_root="test_het_3D", task_id="model", output_dir="out")
model.pcs.add_block(
    # set the process type
    ...)```

# generate a radial 3D mesh and conductivity field
model.msh.generate("radial", dim=3, angles=64, rad=np.arange(101), z_arr=-np.arange(11))
cond = np.exp(srf.mesh(model.msh))
model.mp4.add(name="conductivity")
model.mp4.add_block(# edit recent mpd file
    MSH_TYPE="GROUNDWATER_FLOW",
    MMP_TYPE="PERMEABILITY",
    DIS_TYPE="ELEMENT",
    DATA=by_id(cond),
)
model.mmp.add_block(# permeability, storage and porosity
    GEOMETRY_DIMENSION=3, PERMEABILITY_DISTRIBUTION=model.mp4.file_name
)
model.gli.generate("radial", dim=3, angles=64, rad_out=100, z_size=-10)
model.gli.add_polyline("pwell", [[0, 0, 0], [0, 0, -10]])
for srf in model.gli.SURFACE_NAMES: # set boundary condition
    model.bc.add_block(
        PCS_TYPE="GROUNDWATER_FLOW",
        PRIMARY_VARIABLE="HEAD",
        GEO_TYPE=\"SURFACE\", srf,
        DIS_TYPE=\"CONSTANT\", 0.0,
    )
model.st.add_block(# set pumping condition at the pumpingwell
    PCS_TYPE="GROUNDWATER_FLOW",
    PRIMARY_VARIABLE="HEAD",
    GEO_TYPE=[\"POLYLINE\", "pwell"],
    DIS_TYPE=\"CONSTANT_NEUMANN\", 1.0e-3,
)
model.num.add_block(# numerical solver
    PCS_TYPE="GROUNDWATER_FLOW",
    LINEAR_SOLVER=[2, 5, 1.0e-14, 1000, 1.0, 100, 4],
)
model.out.add_block(# set the output format
    PCS_TYPE="GROUNDWATER_FLOW",
    NOD_VALUES="HEAD",
    GEO_TYPE="DOMAIN",
    DAT_TYPE="VTK",
)
model.write_input()
success = model.run_model()
model.msh.show(show_cell_data=\"Conductivity\": cond, log_scale=True)
files = model.output_files(pcs="GROUNDWATER_FLOW", typ="VTK")
show_vtk(files[-1], log_scale=True)  # show the last time-step
2.2. Tutorial 2: Interaction with GSTools
2.3 Tutorial 2: Interaction with pygmsh

In this example we are generating different meshes with the aid of `pygmsh`.

```python
import numpy as np
from ogs5py import OGS
import pygmsh

geom = pygmsh.built_in.Geometry()
poly = geom.add_polygon([ 0.0, 0.5, 0.0],
                        [-0.1, 0.1, 0.0],
                        [-0.5, 0.0, 0.0],
                        [-0.1, -0.1, 0.0],
                        [ 0.0, -0.5, 0.0],
                        [ 0.1, -0.1, 0.0],
                        [ 0.5, 0.0, 0.0],
                        [ 0.1, 0.1, 0.0] ],
                        lcar=0.05)
axis = [0, 0, 1]
geom.extrude(poly,
              translation_axis=axis,
              rotation_axis=axis,
              point_on_axis=[0, 0, 0],
              angle=2.0 / 6.0 * np.pi)

model = OGS()  # dummy model
# generate example above
model.msh.generate("gmsh", geo_object=geom)
model.msh.show()
# generate a predefined grid adapter in 2D
model.msh.generate("grid_adapter2D", in_mat=1, out_mat=0, fill=True)
model.msh.show(show_material_id=True)
# generate a predefined grid adapter in 3D
model.msh.generate("grid_adapter3D", in_mat=1, out_mat=0, fill=True)
model.msh.show(show_material_id=True)
# generate a predefined block adapter in 3D
model.msh.generate("block_adapter3D", xy_dim=5.0, z_dim=1.0, in_res=1)
model.msh.show(show_element_id=True)
```
Chapter 2. ogs5py Tutorials
3.1 Purpose

ogs5py is a python-API for the OpenGeoSys 5 scientific modeling package. The following functionalities are directly provided on module-level.

3.2 Subpackages

<table>
<thead>
<tr>
<th>Subpackage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileclasses</td>
<td>ogs5py subpackage providing the file classes.</td>
</tr>
<tr>
<td>reader</td>
<td>ogs5py subpackage providing reader for the ogs5 output.</td>
</tr>
<tr>
<td>tools</td>
<td>ogs5py subpackage providing tools.</td>
</tr>
</tbody>
</table>

3.3 Classes

OGS model Base Class

Class to setup an ogs model

\[ \text{OGS}([\text{task\_root}, \text{task\_id}, \text{output\_dir}]) \] 

Class for an OGS5 model.

File Classes

Classes for all OGS5 Files. See: \texttt{ogs5py.fileclasses}

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASC(\texttt{OGS_Config})</td>
<td>Class for the ogsASC file.</td>
</tr>
<tr>
<td>BCI(\texttt{OGS_Config})</td>
<td>Class for the ogs BOUNDARY CONDITION file.</td>
</tr>
<tr>
<td>CCT(\texttt{OGS_Config})</td>
<td>Class for the ogs COMMUNICATION TABLE file.</td>
</tr>
<tr>
<td>DDC(\texttt{OGS_Config})</td>
<td>Class for the ogs MPI DOMAIN DECOMPOSITION file.</td>
</tr>
<tr>
<td>FCT(\texttt{OGS_Config})</td>
<td>Class for the ogs FUNCTION file.</td>
</tr>
</tbody>
</table>
3.4 Functions

Geometric

Geometric routines

\[ \text{hull_deform}(x_{\text{in}}, y_{\text{in}}, z_{\text{in}}, \ldots) \] Providing a transformation function to deform a given mesh.

Searching

Routine to search for a valid ogs id in a directory

\[ \text{search_task_id}(\text{task_root}, \text{search_ext}) \] Search for OGS model names in the given path.

Formatting

Routines to format/generate data in the right way for the input
by_id(array[, ids])
Return a flattened array side-by-side with the array-element ids.

specialrange(val_min, val_max, steps[, typ])
Calculation of special point ranges.

generate_time(time_array[, time_start, ...])
Return a dictionary for the "*.tim" file.

**Downloading**
Routine to download OGS5.

download_ogs([version, system, path, name, ...])
Download the OGS5 executable.

add_exe(ogs_exe[, dest_name])
Add an OGS5 exe to OGS5PY_CONFIG.

reset_download()
Reset all downloads in OGS5PY_CONFIG.

OGS5PY_CONFIG
Standard config path for ogs5py.

**Plotting**
Routine to download OGS5.

show_vtk(vtkfile[, log_scale])
Display a given mesh colored by its material ID.

**Information**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OGS_EXT</td>
<td>all ogs file extensions</td>
</tr>
<tr>
<td>PCS_TYP</td>
<td>PCS types</td>
</tr>
<tr>
<td>PRIM_VAR_BY_PCS</td>
<td>primary variables per PCS</td>
</tr>
</tbody>
</table>
3.5 ogs5py.ogs

Base Class for an OGS5 run.

**OGS Class**

```python
class OGS((task_root, task_id, output_dir))
```

Class for an OGS5 model.

In this class everything for an OGS5 model can be specified.

**Parameters**

- `task_root` *(str, optional)* – Path to the destiny model folder. Default: `cwd+"ogs5model"`
- `task_id` *(str, optional)* – Name for the ogs task. Default: “model”
- `output_dir` *(str or None, optional)* – Path to the output directory. Default: None

**Notes**

The following Classes are present as attributes

- `bc` [Boundary Condition] Information of the Boundary Conditions for the model.
- `cct` [Communication Table] Information of the Communication Table for the model.
- `fct` [Function] Information of the Function definitions for the model.
- `gem` [geochemical thermodynamic modeling coupling] Information of the geochemical thermodynamic modeling coupling for the model.
- `gli` [Geometry] Information of the Geometry for the model.
- `ic` [Initial Condition] Information of the Initial Conditions for the model.
- `krc` [Kinetic Reaction] Information of the Kinetic Reaction for the model.
- `mcp` [reactive components for modelling chemical processes] Information of the reactive components for modelling chemical processes for the model.
- `mfp` [Fluid Properties] Information of the Fluid Properties for the model.
- `mmp` [Medium Properties] Information of the Medium Properties for the model.
- `msh` [Mesh] Information of the Mesh for the model.
- `msp` [Solid Properties] Information of the Solid Properties for the model.
- `num` [Settings for the numerical solver] Information of the numerical solver for the model.
- `out` [Output Settings] Information of the Output Settings for the model.
- `pcs` [Process settings] Information of the Process settings for the model.
- `pct` [Particle Definition for Random walk] Information of the Particles defined for Randomwalk setting.
- `pqc` [Phreqqc coupling (not supported yet)] Information of the Boundary Conditions for the model.
pqcdat [Phreqc coupling (the phreeqc.dat file)] phreeqc.dat file for the model. (just a line-wise file with no comfort)

rei [Reaction Interface] Information of the Reaction Interface for the model.

rfd [definition of time-curves for varying BCs or STs] Information of the time curves for the model.

st [Source Term] Information of the Source Term for the model.

tim [Time settings] Information of the Time settings for the model.

Additional

mpd [Distributed Properties (list of files)] Information of the Distributed Properties for the model.

gli_ext [list for external Geometry definition] External definition of surfaces (TIN) or polylines (POINT_VECTOR)

rfr [list of restart files] RESTART files as defined in the INITIAL_CONDITION

gem_init [list of GEMS3K input files (lst file)] given as GEMinit classes

asc [list of ogs ASC files] This file type comes either from .tim .pcs or .gem

copy_files [list of path-strings] Files that should be copied to the destiny folder.

Attributes

bot_com Get and set the bottom comment for the ogs files.

has_output_dir bool: State if the model has a output directory.

output_dir str: output directory path of the ogs model.

task_id str: task_id (name) of the ogs model.

task_root Get and set the task_root path of the ogs model.

top_com Get and set the top comment for the ogs files.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_asc(self, asc_file)</td>
<td>Method to add a ASC file.</td>
</tr>
<tr>
<td>add_copy_file(self, path)</td>
<td>Method to add an arbitrary file that should be copied.</td>
</tr>
<tr>
<td>add_gem_init(self, gem_init_file)</td>
<td>Method to add a GEMS3K input file.</td>
</tr>
<tr>
<td>add_gli_ext(self, gli_ext_file)</td>
<td>Method to add an external Geometry definition file to the model.</td>
</tr>
<tr>
<td>add_mpd(self, mpd_file)</td>
<td>Method to add an ogs MEDIUM_PROPERTIES_DISTRIBUTED file to the model.</td>
</tr>
<tr>
<td>add_rfr(self, rfr_file)</td>
<td>Method to add an ogs RESTART file to the model.</td>
</tr>
<tr>
<td>del_asc(self[, index])</td>
<td>Method to delete a ASC file.</td>
</tr>
<tr>
<td>del_copy_file(self[, index])</td>
<td>Method to delete a copy-file.</td>
</tr>
<tr>
<td>del_gem_init(self[, index])</td>
<td>Method to delete GEMS3K input file.</td>
</tr>
<tr>
<td>del_gli_ext(self[, index])</td>
<td>Method to delete external Geometry file.</td>
</tr>
<tr>
<td>del_mpd(self[, index])</td>
<td>Method to delete MEDIUM_PROPERTIES_DISTRIBUTED file.</td>
</tr>
<tr>
<td>del_rfr(self[, index])</td>
<td>Method to delete RESTART file.</td>
</tr>
<tr>
<td>gen_script(self[, script_dir, script_name, ...])</td>
<td>Generate a python script for the given model.</td>
</tr>
<tr>
<td>load_model(self[, task_root[, task_id, ...]])</td>
<td>Load an existing OGS5 model.</td>
</tr>
<tr>
<td>output_files(self[, pcs, typ, element, ...])</td>
<td>Get a list of output file paths.</td>
</tr>
</tbody>
</table>
Table 11 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>readpvd(self[, pcs, output_dir])</code></td>
<td>Read the paraview pvd files of this OGS5 model.</td>
</tr>
<tr>
<td><code>readtec_point(self[, pcs, output_dir])</code></td>
<td>Collect TECPLOT point output from this OGS5 model.</td>
</tr>
<tr>
<td><code>readtec_polyline(self[, pcs, trim, output_dir])</code></td>
<td>Collect TECPLOT polyline output from this OGS5 model.</td>
</tr>
<tr>
<td><code>readvtk(self[, pcs, output_dir])</code></td>
<td>Reader for vtk outputfiles of this OGS5 model.</td>
</tr>
<tr>
<td><code>reset(self)</code></td>
<td>Delete every content.</td>
</tr>
<tr>
<td><code>run_model(self[, ogs_exe, ogs_name, ...])</code></td>
<td>Run the defined OGS5 model.</td>
</tr>
<tr>
<td><code>write_input(self)</code></td>
<td>Method to call all write_file() methods that are initialized.</td>
</tr>
</tbody>
</table>

**add_asc (self, asc_file)**

Method to add a ASC file.

See ogs5py.ASC for further information

**add_copy_file (self, path)**

Method to add an arbitrary file that should be copied.

The base-name of the file will be kept and it will be copied to the task-root when the “write” routine is called.

**add_gem_init (self, gem_init_file)**

Method to add a GEMS3K input file.

This is usually generated by GEM-SELEKTOR.

See ogs5py.GEM and ogs5py.GEMinit for further information

**add_gli_ext (self, gli_ext_file)**

Method to add an external Geometry definition file to the model.

This is used for TIN definition in SURFACE or POINT VECTOR definition in POLYLINE in the GLI file.

See ogs5py.GLI for further information

**add_mpd (self, mpd_file)**

Method to add an ogs MEDIUM_PROPERTIES_DISTRIBUTED file to the model.

This is used for distributed information in the MMP file.

See ogs5py.MPD for further information

**add_rfr (self, rfr_file)**

Method to add an ogs RESTART file to the model.

This is used for distributed information in the IC file.

See ogs5py.IC for further information

**del_asc (self, index=None)**

Method to delete a ASC file.

Parameters:

- `index` (int or None, optional) – The index of the ASC file that should be deleted. If None, all ASC files are deleted. Default: None

**del_copy_file (self, index=None)**

Method to delete a copy-file.

Parameters:

- `index` (int or None, optional) – The index of the copy-file that should be deleted. If None, all copy-files are deleted. Default: None

**del_gem_init (self, index=None)**

Method to delete GEMS3K input file.
del_gli_ext (self, index=None)

Method to delete external Geometry file.

Parameters

- **index (int or None, optional)** – The index of the external gli file that should be deleted. If None, all external gli files are deleted. Default: None

del_mpd (self, index=None)

Method to delete MEDIUM_PROPERTIES_DISTRIBUTED file.

Parameters

- **index (int or None, optional)** – The index of the mpd-file that should be deleted. If None, all mpd-files are deleted. Default: None

del_rfr (self, index=None)

Method to delete RESTART file.

Parameters

- **index (int or None, optional)** – The index of the RESTART file that should be deleted. If None, all RESTART files are deleted. Default: None

gen_script (self, script_dir='/home/docs/checkouts/readthedocs.org/user_builds/ogs5py/checkouts/stable/docs/source/ogs_script', script_name='model.py', ogs_cls_name='model', task_root=None, task_id=None, output_dir=None, separate_files=None)

Generate a python script for the given model.

Parameters

- **script_dir (str)** – target directory for the script
- **script_name (str)** – name for the script file (including .py ending)
- **ogs_cls_name (str)** – name of the model in the script
- **task_root (str)** – used task_root in the script
- **task_id (str)** – used task_id in the script
- **output_dir (str)** – used output_dir in the script
- **separate_files (list of str or None)** – list of files, that should be written to separate files and then loaded from the script

Notes

This will only create BlockFiles from the script. GLI and MSH files as well as every other file are stored separately.

load_model (self, task_root, task_id=None, use_task_root=False, use_task_id=False, skip_files=None, skip_ext=None, encoding=None, verbose=False, search_ext=None)

Load an existing OGS5 model.

Parameters

- **task_root (str)** – Path to the destiny folder.
- **task_id (str or None, optional)** – Task ID of the model to load. If None is given, it will be determined by the found files. If multiple possible task_ids were found, the first one in alphabetic order will be used. Default: None
- **use_task_root (Bool, optional)** – State if the given task_root should be used for this model. Default: False
- **use_task_id (Bool, optional)** – State if the given task_id should be used for this model. Default: False
• **skip_files** (*list or None, optional*) – List of file-names, that should not be read. Default: None

• **skip_ext** (*list or None, optional*) – List of file-extensions, that should not be read. Default: None

• **encoding** (*str or None, optional*) – encoding of the given files. If None is given, the system standard is used. Default: None

• **verbose** (*bool, optional*) – Print information of the reading process. Default: False

• **search_ext** (*str*) – OGS extension that should be searched for. Default: “.pcs”

---

**Notes**

This method will search for all known OGS5 file-extensions in the given path (task_root). Additional files from:

• GLI (POINT_VECTOR + TIN)

• MMP (distributed media properties)

• IC (RESTART)

• GEM (GEM3SK init file)

will be read automatically.

If you get an UnicodeDecodeError try loading with:

```python
encoding="ISO-8859-15"
```

---

**output_files** *(self, pcs=None, typ='VTK', element=None, output_dir=None)*

Get a list of output file paths.

**Parameters**

• **pcs** (*string or None, optional*) – specify the PCS type that should be collected. Possible values are:
  
  - None/”” (no PCS_TYPE specified in *.out)
  
  - "NO_PCS"
  
  - "GROUNDWATER_FLOW"
  
  - "LIQUID_FLOW"
  
  - "RICHARDS_FLOW"
  
  - "AIR_FLOW"
  
  - "MULTI_PHASE_FLOW"
  
  - "PS_GLOBAL"
  
  - "HEAT_TRANSPORT"
  
  - "DEFORMATION"
  
  - "MASS_TRANSPORT"
  
  - "OVERLAND_FLOW"
  
  - "FLUID_MOMENTUM"
  
  - "RANDOM_WALK"

Default: None
• **typ** *(string, optional)* – Type of the output ("VTK", "PVD", "TEC_POINT" or "TEC_POLYLINE"). Default: "VTK"

• **element** *(string or None, optional)* – For tecplot output you can specify the name of the output element. (Point-name of Line-name from GLI file) Default: None

**readpvd** *(self, pcs='ALL', output_dir=None)*

Read the paraview pvd files of this OGS5 model.

All concerned files are converted to a dictionary containing their data

**Parameters**

• **pcs** *(string or None, optional)* – specify the PCS type that should be collected

  - None"""" (no PCS_TYPE specified in *.out)
  - "NO_PCS"
  - "GROUNDWATER_FLOW"
  - "LIQUID_FLOW"
  - "RICHARDS_FLOW"
  - "AIR_FLOW"
  - "MULTI_PHASE_FLOW"
  - "PSGLOBAL"
  - "HEAT_TRANSPORT"
  - "DEFORMATION"
  - "MASS_TRANSPORT"
  - "OVERLAND_FLOW"
  - "FLUID_MOMENTUM"
  - "RANDOM_WALK"

You can get a list with all known PCS-types by setting PCS="ALL" Default: "ALL"

• **output_dir** *(any:'None' or :class:'str', optional)* – Sometimes OGS5 doesn’t put the output in the right directory. You can specify a separate output directory here in this case. Default: :any:'None'

**Returns result** – keys are the point names and the items are the data from the corresponding files if pcs="ALL", the output is a dictionary with the PCS-types as keys

**Return type** *dict*

**readtec_point** *(self, pcs='ALL', output_dir=None)*

Collect TECPLLOT point output from this OGS5 model.

**Parameters**

• **pcs** *(string or None, optional)* – specify the PCS type that should be collected

  - None"""" (no PCS_TYPE specified in *.out)
  - "NO_PCS"
  - "GROUNDWATER_FLOW"
  - "LIQUID_FLOW"
  - "RICHARDS_FLOW"
You can get a list with all known PCS-types by setting pcs="ALL". Default: "ALL"

- **output_dir** (:any:`None` or :class:`str`, optional) – Sometimes OGS5 doesn’t put the output in the right directory. You can specify a separate output directory here in this case. Default: :any:`None`

**Returns result** – Keys are the point names and the items are the data from the corresponding files. If pcs="ALL", the output is a dictionary with the PCS-types as keys.

**Return type**  dict

### readtec_polyline

<table>
<thead>
<tr>
<th>Function</th>
<th>Read TECPLOT polyline output from this OGS5 model.</th>
</tr>
</thead>
</table>

**Parameters**

- **pcs** (string or None, optional) – specify the PCS type that should be collected. Possible values are:
  - None/"" (no PCS_TYPE specified in *.out)
  - "NO_PCS"
  - "GROUNDWATER_FLOW"
  - "LIQUID_FLOW"
  - "RICHARDS_FLOW"
  - "AIR_FLOW"
  - "MULTI_PHASE_FLOW"
  - "PS_GLOBAL"
  - "HEAT_TRANSPORT"
  - "DEFORMATION"
  - "MASS_TRANSPORT"
  - "OVERLAND_FLOW"
  - "FLUID_MOMENTUM"
  - "RANDOM_WALK"

You can get a list with all known PCS-types by setting pcs="ALL". Default: "ALL"

- **output_dir** (:any:`None` or :class:`str`, optional) – Sometimes OGS5 doesn’t put the output in the right directory. You can specify a separate output directory here in this case. Default: :any:`None`
• **trim** (*Bool, optional*) – if the ply_ids are not continuous, there will be “None” values in the output list. If trim is “True” these values will be eliminated. If there is just one output for a polyline, the list will be eliminated and the output will be the single dict. Default: True

**Returns result** – keys are the Polyline names and the items are lists sorted by the ply_id (it is assumed, that the ply_ids are continuous, if not, the corresponding list entries are “None”) if pcs=”ALL”, the output is a dictionary with the PCS-types as keys

**Return type**  
dict

**readvtk** (*self, pcs='ALL', output_dir=None*)  
Reader for vtk outputfiles of this OGS5 model.

**Parameters**

• **pcs** (*string or None, optional*) – specify the PCS type that should be collected Possible values are:
  - None/"" (no PCS_TYPE specified in *.out)
  - "NO_PCS"
  - "GROUNDWATER_FLOW"
  - "LIQUID_FLOW"
  - "RICHARDS_FLOW"
  - "AIR_FLOW"
  - "MULTI_PHASE_FLOW"
  - "PS_GLOBAL"
  - "HEAT_TRANSPORT"
  - "DEFORMATION"
  - "MASS_TRANSPORT"
  - "OVERLAND_FLOW"
  - "FLUID_MOMENTUM"
  - "RANDOM_WALK"

You can get a list with all known PCS-types by setting PCS="ALL" Default : None

• **output_dir** (:any:'None' or :class:'str', optional) – Sometimes OGS5 doesn’t put the output in the right directory. You can specify a separate output directory here in this case. Default: :any:'None'

**Returns result** – keys are the point names and the items are the data from the corresponding files if pcs="ALL", the output is a dictionary with the PCS-types as keys

**Return type**  
dict

**reset** (*self*)  
Delete every content.

**run_model** (*self, ogs_exe=None, ogs_name='ogs', print_log=True, save_log=True, log_path=None, log_name=None, timeout=None*)  
Run the defined OGS5 model.

**Parameters**

• **ogs_exe** (*str or None, optional*) – path to the ogs executable. If None is given, the default sys path will be searched with which. Can be a folder containing the exe with basename: ogs_name. It will first look in the **OGS5PY_CONFIG** folder. Default: None
• **ogs_name** *(str or None, optional)* – Name of the ogs executable to search for. Just used if `ogs_exe` is None. Default: "ogs"

• **print_log** *(bool, optional)* – state if the ogs output should be displayed in the terminal. Default: True

• **save_log** *(bool, optional)* – state if the ogs output should be saved to a file. Default: True

• **log_path** *(str or None, optional)* – Path, where the log file should be saved. Default: None (the defined output directory or the task_root directory)

• **log_name** *(str or None, optional)* – Name of the log file. Default: None (task_id+time+"_log.txt")

• **timeout** *(int or None, optional)* – Time to wait for OGS5 to finish in seconds. Default: None

**Returns** success – State if OGS5 terminated ‘normally’. (Allways true on Windows.)

**Return type** bool

**write_input** *(self)*

Method to call all `write_file()` methods that are initialized.

**bot_com**

Get and set the bottom comment for the ogs files.

**has_output_dir**

State if the model has a output directory.

**Type** bool

**output_dir**

output directory path of the ogs model.

**Type** str

**task_id**

`task_id` (name) of the ogs model.

**Type** str

**task_root**

Get and set the task_root path of the ogs model.

**top_com**

Get and set the top comment for the ogs files.
3.6 ogs5py.fileclasses

ogs5py subpackage providing the file classes.

**Subpackages**

<table>
<thead>
<tr>
<th>Subpackage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>Base Classes for the OGS Files.</td>
</tr>
<tr>
<td>gli</td>
<td>Class for the ogs GEOMETRY file.</td>
</tr>
<tr>
<td>msh</td>
<td>Class for the ogs MESH file.</td>
</tr>
</tbody>
</table>

**File Classes**

Classes for all OGS5 Files

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asc.ASC(<strong>OGS_Config)</strong></td>
<td>Class for the ogs ASC file.</td>
</tr>
<tr>
<td>bc.Bc(<strong>OGS_Config)</strong></td>
<td>Class for the ogs BOUNDARY CONDITION file.</td>
</tr>
<tr>
<td>cct.CCT(<strong>OGS_Config)</strong></td>
<td>Class for the ogs COMMUNICATION TABLE file.</td>
</tr>
<tr>
<td>ddc.DDC(<strong>OGS_Config)</strong></td>
<td>Class for the ogs MPI DOMAIN DECOMPOSITION file.</td>
</tr>
<tr>
<td>fct.FCT(<strong>OGS_Config)</strong></td>
<td>Class for the ogs FUNCTION file.</td>
</tr>
<tr>
<td>gem.GEM(<strong>OGS_Config)</strong></td>
<td>Class for the ogs GEOCHEMICAL THERMODYNAMIC MODELING COUPLING file.</td>
</tr>
<tr>
<td>gem.GEMinit([lst_name, dch, ipm, dbr, ...])</td>
<td>Class for GEMS3K input file.</td>
</tr>
<tr>
<td>gli.GLI([gli_dict])</td>
<td>Class for the ogs GEOMETRY file.</td>
</tr>
<tr>
<td>gli.GLIext([typ, data, name, file_ext, ...])</td>
<td>Class for an external definition for the ogs GEOMETRY file.</td>
</tr>
<tr>
<td>ic.IC(<strong>OGS_Config)</strong></td>
<td>Class for the ogs INITIAL_CONDITION file.</td>
</tr>
<tr>
<td>ic.RFR([variables, data, units, headers, ...])</td>
<td>Class for the ogs RESTART file, if the DIS_TYPE in IC is set to RESTART.</td>
</tr>
<tr>
<td>krc.KRC(<strong>OGS_Config)</strong></td>
<td>Class for the ogs KINETRIC REACTION file.</td>
</tr>
<tr>
<td>mcp.MCP(<strong>OGS_Config)</strong></td>
<td>Class for the ogs COMPONENT_PROPERTIES file.</td>
</tr>
<tr>
<td>mmp.MMP(<strong>OGS_Config)</strong></td>
<td>Class for the ogs MEDIUM_PROPERTIES file.</td>
</tr>
<tr>
<td>mpd.MPD([name, file_ext])</td>
<td>Class for the ogs MEDIUM_PROPERTIES DISTRIBUTED file.</td>
</tr>
<tr>
<td>msh.MSH([mesh_list])</td>
<td>Class for a multi layer mesh file that contains multiple #FEM_MSH Blocks.</td>
</tr>
<tr>
<td>msp.MSP(<strong>OGS_Config)</strong></td>
<td>Class for the ogs SOLID_PROPERTIES file.</td>
</tr>
<tr>
<td>num.NUM(<strong>OGS_Config)</strong></td>
<td>Class for the ogs NUMERICS file.</td>
</tr>
<tr>
<td>out.OUT(<strong>OGS_Config)</strong></td>
<td>Class for the ogs OUTPUT file.</td>
</tr>
<tr>
<td>pcs.PCS(<strong>OGS_Config)</strong></td>
<td>Class for the ogs PROCESS file.</td>
</tr>
<tr>
<td>pct.PCT([data, s_flag, task_root, task_id])</td>
<td>Class for the ogs Particle file, if the PCS TYPE is RANDOM_WALK.</td>
</tr>
<tr>
<td>pqc.PQC(<strong>OGS_Config)</strong></td>
<td>Class for the ogs PHREEQC interface file.</td>
</tr>
<tr>
<td>pqc.PQCDat(<strong>OGS_Config)</strong></td>
<td>Class for the ogs PHREEQC dat file.</td>
</tr>
<tr>
<td>rei.REI(<strong>OGS_Config)</strong></td>
<td>Class for the ogs REACTION_INTERFACE file.</td>
</tr>
<tr>
<td>rfd.RFD(<strong>OGS_Config)</strong></td>
<td>Class for the ogs USER DEFINED TIME CURVES file.</td>
</tr>
<tr>
<td>st.St(<strong>OGS_Config)</strong></td>
<td>Class for the ogs SOURCE_TERM file.</td>
</tr>
<tr>
<td>tim.TIM(<strong>OGS_Config)</strong></td>
<td>Class for the ogs TIME_STEPPING file.</td>
</tr>
</tbody>
</table>
ogs5py/fileclasses/base

Base Classes for the OGS Files.

File Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>File class with minimal functionality.</td>
</tr>
<tr>
<td>LineFile</td>
<td>OGS class to handle line-wise text files.</td>
</tr>
<tr>
<td>BlockFile</td>
<td>OGS Base class to derive all file formats.</td>
</tr>
<tr>
<td>MultiFile</td>
<td>Class holding multiple files of the same type.</td>
</tr>
</tbody>
</table>

```python
class BlockFile(task_root=None, task_id='model', file_ext='.std')

Bases: ogs5py.fileclasses.base.File

OGS Base class to derive all file formats.

Parameters

- **task_root** *(str, optional)* – Path to the destiny model folder. Default: `cwd+'ogs5model'`
- **task_id** *(str, optional)* – Name for the ogs task. Default: “model”
- **file_ext** *(str, optional)* – extension of the file (with leading dot “.std”) Default: “.std”

Attributes

- **block_no** Number of blocks in the file.
- **file_name** str: base name of the file with extension.
- **file_path** str: save path of the file.
- **force_writing** bool: state if the file is written even if empty.
- **is_empty** State if the OGS file is empty.
- **name** str: name of the file without extension.

Methods

- **add_block** *(self[, index, main_key])* Add a new Block to the actual file.
- **add_content** *(self, content[, main_index, ...])* Add single-line content to the actual file.
- **add_copy_link** *(self, path[, symlink])* Add a link to copy a file instead of writing.
- **add_main_keyword** *(self, key[, main_index])* Add a new main keyword (#key) to the actual file.
- **add_multi_content** *(self, content[, ...])* Add multiple content to the actual file.
- **add_sub_keyword** *(self[, key[, main_index, ...]])* Add a new sub keyword ($key) to the actual file.
- **append_to_block** *(self[, index])* Append data to an existing Block in the actual file.
- **check** *(self[, verbose])* Check if the given file is valid.
- **del_block** *(self[, index, del_all])* Delete a block by its index.
- **del_content** *(self[, main_index, sub_index, ...])* Delete content by its position.
- **del_copy_link** *(self)* Remove a former given link to an external file.
- **del_main_keyword** *(self[, main_index, del_all])* Delete a main keyword (#key) by its position.
- **del_sub_keyword** *(self[, main_index, ...])* Delete a sub keyword ($key) by its position.
```

Continued on next page
Table 15 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_block</code></td>
<td>Get a Block from the actual file.</td>
</tr>
<tr>
<td><code>get_block_no</code></td>
<td>Get the number of blocks in the file.</td>
</tr>
<tr>
<td><code>get_file_type</code></td>
<td>Get the OGS file class name.</td>
</tr>
<tr>
<td><code>get_multi_keys</code></td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td><code>is_block_unique</code></td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td><code>read_file</code></td>
<td>Read an existing OGS input file.</td>
</tr>
<tr>
<td><code>reset</code></td>
<td>Delete every content.</td>
</tr>
<tr>
<td><code>save</code></td>
<td>Save the actual OGS input file in the given path.</td>
</tr>
<tr>
<td><code>update_block</code></td>
<td>Update a Block from the actual file.</td>
</tr>
<tr>
<td><code>write_file</code></td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>

```
add_block (self, index=None, main_key=None, **block)
```

Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1 content1 . . .
$SUBKEY2 content2 . . .
```

which looks like the following:

```
FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- `index (int or None, optional)` – Positional index, where to insert the given block. As default, it will be added at the end. Default: None.
- `main_key (string, optional)` – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type
- `**block (keyword dict)` – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

```
add_content (self, content, main_index=None, sub_index=None, line_index=None)
```

Add single-line content to the actual file.

**Parameters**

- `content (list)` – list containing one line of content given as a list of single statements
- `main_index (int, optional)` – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- `sub_index (int, optional)` – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- `line_index (int, optional)` – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

3.6. ogs5py.fileclasses
**add_copy_link** *(self, path, symlink=False)*

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

Parameters

- **path** *(str)* – path to the existing file that should be copied
- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword** *(self, key, main_index=None)*

Add a new main keyword (#key) to the actual file.

Parameters

- **key** *(string)* – key name
- **main_index** *(int, optional)* – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

**add_multi_content** *(self, content, main_index=None, sub_index=None)*

Add multiple content to the actual file.

Parameters

- **content** *(list)* – list containing lines of content, each given as a list of single statements
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

Notes

There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

**add_sub_keyword** *(self, key, main_index=None, sub_index=None)*

Add a new sub keyword ($key) to the actual file.

Parameters

- **key** *(string)* – key name
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

Notes

There needs to be at least one main keyword, otherwise the subkeyword is not added.

**append_to_block** *(self, index=None, **block)*

Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:
#MAIN_KEY

$SUBKEY1  content1 …

$SUBKEY2  content2 …

which looks like the following:

FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)

Parameters

- **index** (*int or None, optional*) — Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

- **block** (*keyword dict*) — here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content.

**check**(self, verbose=True)
Check if the given file is valid.

Parameters **verbose** (*bool, optional*) — Print information for the executed checks.
Default: True

Returns **result** — Validity of the given file.

Return type **bool**

**del_block**(self, index=None, del_all=False)
Delete a block by its index.

Parameters

- **index** (*int or None, optional*) — Positional index of the block of interest. As default, the last one is returned. Default: None

- **del_all** (*bool, optional*) — State, if all blocks shall be deleted. Default: False

**del_content**(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)
Delete content by its position.

Parameters

- **main_index** (*int, optional*) — index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

- **sub_index** (*int, optional*) — index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.

- **line_index** (*int, optional*) — position of the content line, that should be deleted. Default: -1

- **del_all** (*bool, optional*) — State, if all content shall be deleted. Default: False

**del_copy_link**(self)
Remove a former given link to an external file.

**del_main_keyword**(self, main_index=None, del_all=False)
Delete a main keyword (#key) by its position.

Parameters

- **main_index** (*int, optional*) — position, which main keyword should be deleted. Default: -1

- **del_all** (*bool, optional*) — State, if all main keywords shall be deleted. Default: False
del_sub_keyword(self, main_index=-1, sub_index=-1, del_all=False)

Delete a sub keyword ($key) by its position.

Parameters

- **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- **pos** (*int, optional*) – position, which sub keyword should be deleted. Default: -1
- **del_all** (*bool, optional*) – State, if all sub keywords shall be deleted. Default: False

get_block(self, index=None, as_dict=True)

Get a Block from the actual file.

Parameters

- **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is returned. Default: None
- **as_dict** (*bool, optional*) – Here you can state if you want the output as a dictionary, which can be used as key-word-arguments for add_block. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

get_block_no(self)

Get the number of blocks in the file.

get_file_type(self)

Get the OGS file class name.

get_multi_keys(self, index=None)

State if a block has a unique set of sub keywords.

is_block_unique(self, index=None)

State if a block has a unique set of sub keywords.

read_file(self, path, encoding=None, verbose=False)

Read an existing OGS input file.

Parameters

- **path** (*str*) – path to the existing file that should be read
- **encoding** (*str or None, optional*) – encoding of the given file. If None is given, the system standard is used. Default: None
- **verbose** (*bool, optional*) – Print information of the reading process. Default: False

reset(self)

Delete every content.

save(self, path, **kwargs)

Save the actual OGS input file in the given path.

Parameters

- **path** (*str*) – path to where file should be saved
- **update** (*bool, optional*) – state if the content should be updated before saving. Default: True

update_block(self, index=None, main_key=None, **block)

Update a Block from the actual file.

Parameters

- **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is used. Default: None
• **main_key** (*string, optional*) – Main keyword of the block that should be updated (see: `MKEYS`). This shouldn’t be done. Default: None
• **block** (*keyword dict*) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: `SUBKEY=content`

```python
write_file(self)
Write the actual OGS input file to the given folder.
Its path is given by “task_root+task_id+file_ext”.
```

```python
MKEYS = []
Main Keywords of this OGS-BlockFile
```

```python
SKEYS = []
Sub Keywords of this OGS-BlockFile
```

```python
STD = {}
Standard Block OGS-BlockFile
```

```python
block_no
Number of blocks in the file.
```

```python
file_name
base name of the file with extension.
```

```python
file_path
save path of the file.
```

```python
force_writing
state if the file is written even if empty.
```

```python
is_empty
State if the OGS file is empty.
```

```python
name
name of the file without extension.
```

```python
class File(task_root=None, task_id='model', file_ext='.std')
Bases: object
File class with minimal functionality.
```

```python
Parameters
• **task_root** (*str, optional*) – Path to the destiny folder. Default is cwd+”ogs5model”
• **task_id** (*str, optional*) – Name for the ogs task. Default: “model”
• **file_ext** (*str, optional*) – extension of the file (with leading dot “.std”) Default: “.std”
```

```python
Attributes

  file_name *str*: base name of the file with extension.
```
**file_path** str: save path of the file.

**force_writing** bool: state if the file is written even if empty.

**is_empty** bool: state if the OGS file is empty.

**name** str: name of the file without extension.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_copy_link</td>
<td>Add a link to copy a file instead of writing.</td>
</tr>
<tr>
<td>check</td>
<td>Check if the given file is valid.</td>
</tr>
<tr>
<td>del_copy_link</td>
<td>Remove a former given link to an external file.</td>
</tr>
<tr>
<td>get_file_type</td>
<td>Get the OGS file class name.</td>
</tr>
<tr>
<td>read_file</td>
<td>Read an existing file.</td>
</tr>
<tr>
<td>reset</td>
<td>Delete every content.</td>
</tr>
<tr>
<td>save</td>
<td>Save the actual file in the given path.</td>
</tr>
<tr>
<td>write_file</td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>

#### add_copy_link(self, path[, symlink=False])

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linkedin to the target folder.

**Parameters**

- `path` (str) – path to the existing file that should be copied
- `symlink` (bool, optional) – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

#### check(self, verbose=True)

Check if the given file is valid.

**Parameters**

- `verbose` (bool, optional) – Print information for the executed checks. Default: True

**Returns**

- `result` (bool) – Validity of the given file.

**Return type**

- `bool`

#### del_copy_link(self)

Remove a former given link to an external file.

#### get_file_type(self)

Get the OGS file class name.

#### read_file(self, path[, encoding, verbose=False])

Read an existing file.

#### reset(self)

Delete every content.

#### save(self, path, **kwargs)

Save the actual file in the given path.

#### write_file(self)

Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

**file_name**

base name of the file with extension.
Type **str**

**file_path**
*save path of the file.*

Type **str**

**force_writing**
*state if the file is written even if empty.*

Type **bool**

**is_empty**
*state if the OGS file is empty.*

Type **bool**

**name**
*name of the file without extension.*

Type **str**

**class LineFile**

```python
class LineFile:
    def __init__(self, lines=None, name=None, file_ext='.txt', task_root=None, task_id='model'):
        pass
```

Bases: **ogs5py.fileclasses.base.File**

OGS class to handle line-wise text files.

**Parameters**

* **lines** *(list of str, optional) – content of the file as a list of lines Default: None*
* **name** *(str, optional) – name of the file without extension Default: “textfile”*
* **file_ext** *(str, optional) – extension of the file (with leading dot “.txt”) Default: “.txt”*
* **task_root** *(str, optional) – Path to the destiny model folder. Default: cwd+"ogs5model"*
* **task_id** *(str, optional) – Name for the ogs task. (a place holder) Default: “model”*

**Attributes**

* **file_name** *(str) – base name of the file with extension.*
* **file_path** *(str) – save path of the file.*
* **force_writing** *(bool) – state if the file is written even if empty.*
* **is_empty** *(bool) – state if the file is empty.*
* **name** *(str) – name of the file without extension.*

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_copy_link</td>
<td>Add a link to copy a file instead of writing.</td>
</tr>
<tr>
<td>check</td>
<td>Check if the given text-file is valid.</td>
</tr>
<tr>
<td>del_copy_link</td>
<td>Remove a former given link to an external file.</td>
</tr>
<tr>
<td>get_file_type</td>
<td>Get the OGS file class name.</td>
</tr>
<tr>
<td>read_file</td>
<td>Read an existing OGS input file.</td>
</tr>
<tr>
<td>reset</td>
<td>Delete every content.</td>
</tr>
<tr>
<td>save</td>
<td>Save the actual line-wise file in the given path.</td>
</tr>
<tr>
<td>write_file</td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>
**add_copy_link** *(self, path, symlink=False)*
Add a link to copy a file instead of writing.
Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

**Parameters**
- **path** *(str)* – path to the existing file that should be copied
- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**check** *(self, verbose=True)*
Check if the given text-file is valid.

**Parameters**
- **verbose** *(bool, optional)* – Print information for the executed checks. Default: True

**Returns**
- **result** – Validity of the given file.

**Return type**
*bool*

**del_copy_link** *(self)*
Remove a former given link to an external file.

**get_file_type** *(self)*
Get the OGS file class name.

**read_file** *(self, path, encoding=None, verbose=False)*
Read an existing OGS input file.

**Parameters**
- **path** *(str)* – path to the existing file that should be read
- **encoding** *(str or None, optional)* – encoding of the given file. If None is given, the system standard is used. Default: None
- **verbose** *(bool, optional)* – Print information of the reading process. Default: False

**reset** *(self)*
Delete every content.

**save** *(self, path)*
Save the actual line-wise file in the given path.

**Parameters**
- **path** *(str)* – path to where to file should be saved

**write_file** *(self)*
Write the actual OGS input file to the given folder.
Its path is given by “task_root+task_id+file_ext”.

**file_name**
- base name of the file with extension.
  
  **Type**
  *str*

**file_path**
- save path of the file.
  
  **Type**
  *str*

**force_writing**
- state if the file is written even if empty.
  
  **Type**
  *bool*

**is_empty**
- state if the file is empty.
Type `bool`

**name**

name of the file without extension.

Type `str`

class **MultiFile** (*base*, **standard**)

Bases: `object`

Class holding multiple files of the same type.

**Parameters**

- *base* (*object*) – Base class for the files
- **standard** – Standard keyword arguments for new instances of the Base class.

**Attributes**

- **id** *int*: The current id to access in the file list.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(self, *args, **kwargs)</code></td>
<td>Add a new instance of the base class with *args and **kwargs.</td>
</tr>
<tr>
<td><code>append(self, file)</code></td>
<td>Append a new file to the list.</td>
</tr>
<tr>
<td><code>delete(self[, file_id])</code></td>
<td>Delete a certain file.</td>
</tr>
<tr>
<td><code>reset_all(self)</code></td>
<td>Reset the Multi File.</td>
</tr>
</tbody>
</table>

`add (self, *args, **kwargs)`

Add a new instance of the base class with *args and **kwargs.

`append (self, file)`

Append a new file to the list.

`delete (self[, file_id=-1])`

Delete a certain file.

`reset_all (self)`

Reset the Multi File.

**id**

The current id to access in the file list.

Type `int`
ogs5py.fileclasses.asc

Class for the ogs ASC file.

File Class

```python
ASC(**OGS_Config)  Class for the ogs ASC file.
```

class ASC(**OGS_Config):
    Bases: ogs5py.fileclasses.base.LineFile
    Class for the ogs ASC file.

Parameters

- **lines**(list of str, optional) – content of the file as a list of lines Default: None
- **name**(str, optional) – name of the file without extension Default: “textfile”
- **task_root**(str, optional) – Path to the destiny folder. Default: cwd+“ogs5model”
- **task_id**(str, optional) – Name for the ogs task. (a place holder) Default: “model”

Notes

This is just handled as a line-wise file. You can access the data by line with:

```python
ASC.lines
```

This file type comes either from .tim .pcs or .gem

Attributes

- **file_name** str: base name of the file with extension.
- **file_path** str: save path of the file.
- **force_writing** bool: state if the file is written even if empty.
- **is_empty** bool: state if the file is empty.
- **name** str: name of the file without extension.

Methods

- **add_copy_link**(self, path[, symlink]) Add a link to copy a file instead of writing.
- **check**(self[, verbose]) Check if the given text-file is valid.
- **del_copy_link**(self) Remove a former given link to an external file.
- **get_file_type**(self) Get the OGS file class name.
- **read_file**(self, path[, encoding, verbose]) Read an existing OGS input file.
- **reset**(self) Delete every content.
- **save**(self, path) Save the actual line-wise file in the given path.
- **write_file**(self) Write the actual OGS input file to the given folder.
add_copy_link (self, path, symlink=False)
Add a link to copy a file instead of writing.
Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target
cfolder.

Parameters

• path (str) – path to the existing file that should be copied
• symlink (bool, optional) – on UNIX systems it is possible to use a symbolic
  link to save time if the file is big. Default: False

check (self, verbose=True)
Check if the given text-file is valid.

Parameters verbose (bool, optional) – Print information for the executed checks.
  Default: True

Returns result – Validity of the given file.

Return type bool
del_copy_link (self)
Remove a former given link to an external file.

get_file_type (self)
Get the OGS file class name.

read_file (self, path, encoding=None, verbose=False)
Read an existing OGS input file.

Parameters

• path (str) – path to the existing file that should be read
• encoding (str or None, optional) – encoding of the given file. If None is
given, the system standard is used. Default: None
• verbose (bool, optional) – Print information of the reading process. Default: False

reset (self)
Delete every content.

save (self, path)
Save the actual line-wise file in the given path.

Parameters path (str) – path to where to file should be saved

write_file (self)
Write the actual OGS input file to the given folder.
Its path is given by “task_root+task_id+file_ext”.

file_name
base name of the file with extension.

Type str

file_path
save path of the file.

Type str

force_writing
state if the file is written even if empty.

Type bool

is_empty
state if the file is empty.
Type `bool`

**name**

name of the file without extension.

Type `str`
ogs5py.fileclasses.bc

Class for the ogs BOUNDARY CONDITION file.

File Class

class BC(**OGS_Config)
Class for the ogs BOUNDARY CONDITION file.

Parameters

• **task_root** (str, optional) – Path to the destiny model folder. Default: cwd+"ogs5model"
• **task_id** (str, optional) – Name for the ogs task. Default: “model”

Notes

Main-Keywords (#):
• BOUNDARY_CONDITION

Sub-Keywords ($) per Main-Keyword:
• BOUNDARY_CONDITION
  – COMP_NAME
  – CONSTRAINED
  – COPY_VALUE
  – DIS_TYPE
  – DIS_TYPE_CONDITION
  – EPSILON
  – EXCAVATION
  – FCT_TYPE
  – GEO_TYPE
  – MSH_TYPE
  – NO_DISP_INCREMENT
  – PCS_TYPE
  – PRESSURE_AS_HEAD
  – PRIMARY_VARIABLE
  – TIME_CONTROLLED_ACTIVE
  – TIME_INTERVAL
  – TIM_TYPE

Standard block:

PCS_TYPE “GROUNDWATER_FLOW”
PRIMARY VARIABLE "HEAD"

DIS_TYPE ["CONSTANT", 0.0]

GEO_TYPE ["POLYLINE", "boundary"]


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rf_bc_new.cpp#L228

See also:

add_block

Attributes

block_no Number of blocks in the file.

file_name str: base name of the file with extension.

file_path str: save path of the file.

force_writing bool: state if the file is written even if empty.

is_empty State if the OGS file is empty.

name str: name of the file without extension.

Methods

add_block(self[, index, main_key]) Add a new Block to the actual file.

add_content(self, content[, main_index, ...]) Add single-line content to the actual file.

add_copy_link(self, path[, symlink]) Add a link to copy a file instead of writing.

add_main_keyword(self, key[, main_index]) Add a new main keyword (#key) to the actual file.

add_multi_content(self, content[, ...]) Add multiple content to the actual file.

add_sub_keyword(self, key[, main_index, ...]) Add a new sub keyword ($key) to the actual file.

append_to_block(self[, index]) Append data to an existing Block in the actual file.

check(self[, verbose]) Check if the given file is valid.

del_block(self[, index, del_all]) Delete a block by its index.

del_content(self[, main_index, sub_index, ...]) Delete content by its position.

del_copy_link(self) Remove a former given link to an external file.

del_main_keyword(self[, main_index, del_all]) Delete a main keyword (#key) by its position.

del_sub_keyword(self[, main_index, ...]) Delete a sub keyword ($key) by its position.

get_block(self[, index, as_dict]) Get a Block from the actual file.

get_block_no(self) Get the number of blocks in the file.

get_file_type(self) Get the OGS file class name.

get_multi_keys(self[, index]) State if a block has a unique set of sub keywords.

is_block_unique(self[, index]) State if a block has a unique set of sub keywords.

read_file(self, path[, encoding, verbose]) Read an existing OGS input file.

reset(self) Delete every content.

save(self, path[, encoding, verbose]) Save the actual OGS input file in the given path.

update_block(self[, index, main_key]) Update a Block from the actual file.

write_file(self) Write the actual OGS input file to the given folder.

add_block (self, index=None, main_key=None, **block)
Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:
#MAIN_KEY

$SUBKEY1 content1 …

$SUBKEY2 content2 …

which looks like the following:

FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)

**Parameters**

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

- **main_key** *(string, optional)* – Main keyword of the block that should be added (see: MKEYS). Default: the first main keyword of the file-type

- ****block**(keyword dict)** – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**add_content** *(self, content, main_index=None, sub_index=None, line_index=None)*

Add single-line content to the actual file.

**Parameters**

- **content** *(list)* – list containing one line of content given as a list of single statements

- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

- **line_index** *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one (""”) will be added and the content is then directly connected to the actual main keyword.

**add_copy_link** *(self, path, symlink=False)*

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

**Parameters**

- **path** *(str)* – path to the existing file that should be copied

- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword** *(self, key, main_index=None)*

Add a new main keyword (#key) to the actual file.

**Parameters**

- **key** *(string)* – key name

- **main_index** *(int, optional)* – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.
add_multi_content\( (\text{self}, \text{content}, \text{main\_index}=\text{None}, \text{sub\_index}=\text{None}) \)

Add multiple content to the actual file.

**Parameters**

- **content** \((\text{list})\) – list containing lines of content, each given as a list of single statements
- **main\_index** \((\text{int, optional})\) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub\_index** \((\text{int, optional})\) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one ("\") will be added and the content is then directly connected to the actual main keyword.

add_sub_keyword\( (\text{self}, \text{key}, \text{main\_index}=\text{None}, \text{sub\_index}=\text{None}) \)

Add a new sub keyword \((\$\text{key})\) to the actual file.

**Parameters**

- **key** \((\text{string})\) – key name
- **main\_index** \((\text{int, optional})\) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub\_index** \((\text{int, optional})\) – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the subkeyword is not added.

append_to_block\( (\text{self}, \text{index}=\text{None}, **\text{block}) \)

Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN\_KEY

$\text{SUBKEY1} \text{content1} \ldots

$\text{SUBKEY2} \text{content2} \ldots
```

which looks like the following:

FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)

**Parameters**

- **index** \((\text{int or None, optional})\) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- ****block** \((\text{keyword dict})\) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

check\( (\text{self}, \text{verbose}=\text{True}) \)

Check if the given file is valid.
Parameters `verbose (bool, optional)` – Print information for the executed checks. Default: True

Returns `result` – Validity of the given file.

Return type `bool`

del_block (self, index=None, del_all=False)
Delete a block by its index.

Parameters

* `index (int or None, optional)` – Positional index of the block of interest. As default, the last one is returned. Default: None

* `del_all (bool, optional)` – State, if all blocks shall be deleted. Default: False

del_content (self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)
Delete content by its position.

Parameters

* `main_index (int, optional)` – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

* `sub_index (int, optional)` – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.

* `line_index (int, optional)` – position of the content line, that should be deleted. Default: -1

* `del_all (bool, optional)` – State, if all content shall be deleted. Default: False

del_copy_link (self)
Remove a former given link to an external file.

del_main_keyword (self, main_index=None, del_all=False)
Delete a main keyword (#key) by its position.

Parameters

* `main_index (int, optional)` – position, which main keyword should be deleted. Default: -1

* `del_all (bool, optional)` – State, if all main keywords shall be deleted. Default: False

del_sub_keyword (self, main_index=-1, sub_index=-1, del_all=False)
Delete a sub keyword ($key) by its position.

Parameters

* `main_index (int, optional)` – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

* `pos (int, optional)` – position, which sub keyword should be deleted. Default: -1

* `del_all (bool, optional)` – State, if all sub keywords shall be deleted. Default: False

get_block (self, index=None, as_dict=True)
Get a Block from the actual file.

Parameters

* `index (int or None, optional)` – Positional index of the block of interest. As default, the last one is returned. Default: None
• `as_dict` *(bool, optional)* – Here you can state if you want the output as a dictionary, which can be used as key-word-arguments for `add_block`. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

`get_block_no` *(self)*
Get the number of blocks in the file.

`get_file_type` *(self)*
Get the OGS file class name.

`get_multi_keys` *(self, index=None)*
State if a block has a unique set of sub keywords.

`is_block_unique` *(self, index=None)*
State if a block has a unique set of sub keywords.

`read_file` *(self, path, encoding=None, verbose=False)*
Read an existing OGS input file.

**Parameters**

• `path` *(str)* – path to the existing file that should be read

• `encoding` *(str or None, optional)* – encoding of the given file. If None is given, the system standard is used. Default: None

• `verbose` *(bool, optional)* – Print information of the reading process. Default: False

`reset` *(self)*
Delete every content.

`save` *(self, path, **kwargs)*
Save the actual OGS input file in the given path.

**Parameters**

• `path` *(str)* – path to where to file should be saved

• `update` *(bool, optional)* – state if the content should be updated before saving. Default: True

`update_block` *(self, index=None, main_key=None, **block)*
Update a Block from the actual file.

**Parameters**

• `index` *(int or None, optional)* – Positional index of the block of interest. As default, the last one is used. Default: None

• `main_key` *(string, optional)* – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None

• `**block` *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

`write_file` *(self)*
Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['BOUNDARY_CONDITION']

SKEYS = [['PCS_TYPE', 'PRIMARY_VARIABLE', 'COMP_NAME', 'GEO_TYPE', 'DIS_TYPE', 'TIM_TYPE', 'FCT_TYPE', 'MSH_TYPE', ... 'TIME_INTERVAL', 'EXCAVATION', 'NO_DISP_INCREMENT', 'COPY_VALUE', 'PRESSURE_AS_HEAD', 'CONSTRAINED']

STD = {'DIS_TYPE': ['CONSTANT', 0.0], 'GEO_TYPE': ['POLYLINE', 'boundary'], 'PCS_TYPE':

block_no
Number of blocks in the file.
file_name
    base name of the file with extension.
    Type str

file_path
    save path of the file.
    Type str

force_writing
    state if the file is written even if empty.
    Type bool

is_empty
    State if the OGS file is empty.

name
    name of the file without extension.
    Type str
ogs5py.fileclasses.cct

Class for the ogs COMMUNICATION TABLE file.

File Class

class CCT(**OGS_Config)

File Class for the ogs COMMUNICATION TABLE file.

class CCT(**OGS_Config)

File Class for the ogs COMMUNICATION TABLE file.

Parameters

- `task_root` *(str, optional)* – Path to the destiny model folder. Default: `cwd+"ogs5model"`
- `task_id` *(str, optional)* – Name for the ogs task. Default: “model”

Notes

Main-Keywords (#):

- COMMUNICATION_TABLE

Sub-Keywords ($) per Main-Keyword:

- COMMUNICATION_TABLE
  - MYRANK
  - NEIGHBOR
  - NNEIGHBORS

Standard block: None


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/fct_mpi.cpp#L27

See also:

`add_block`

Attributes

- `block_no` Number of blocks in the file.
- `file_name` *(str)*: base name of the file with extension.
- `file_path` *(str)*: save path of the file.
- `force_writing` *(bool)*: state if the file is written even if empty.
- `is_empty` State if the OGS file is empty.
- `name` *(str)*: name of the file without extension.

Methods
add_block(self[, index, main_key])
   Add a new Block to the actual file.

add_content(self, content[, main_index, ...])
   Add single-line content to the actual file.

add_copy_link(self, path[, symlink])
   Add a link to copy a file instead of writing.

add_main_keyword(self, key[, main_index])
   Add a new main keyword (#key) to the actual file.

add_multi_content(self, content[, ...])
   Add multiple content to the actual file.

add_sub_keyword(self[, key, main_index, ...])
   Add a new sub keyword ($key) to the actual file.

append_to_block(self[, index])
   Append data to an existing Block in the actual file.

check(self[, verbose])
   Check if the given file is valid.

del_block(self[, index, del_all])
   Delete a block by its index.

del_content(self[, main_index, sub_index, ...])
   Delete content by its position.

del_copy_link(self)
   Remove a former given link to an external file.

del_main_keyword(self[, main_index, del_all])
   Delete a main keyword (#key) by its position.

del_sub_keyword(self[, main_index, ...])
   Delete a sub keyword ($key) by its position.

get_block(self[, index, as_dict])
   Get a Block from the actual file.

get_block_no(self)
   Get the number of blocks in the file.

get_file_type(self)
   Get the OGS file class name.

get_multi_keys(self[, index])
   State if a block has a unique set of sub keywords.

is_block_unique(self[, index])
   State if a block has a unique set of sub keywords.

read_file(self, path[, encoding, verbose])
   Read an existing OGS input file.

reset(self)
   Delete every content.

save(self, path, **kwargs)
   Save the actual OGS input file in the given path.

update_block(self[, index, main_key])
   Update a Block from the actual file.

write_file(self)
   Write the actual OGS input file to the given folder.

add_block (self, index=None, main_key=None, **block)
   Add a new Block to the actual file.

   Keywords are the sub keywords of the actual file type:

   #MAIN_KEY
      $SUBKEY1 content1 ...
      $SUBKEY2 content2 ...

   which looks like the following:

   FILE.add_block (SUBKEY1=content1, SUBKEY2=content2)

Parameters

   - **index**(int or None, optional) – Positional index, where to insert the given
     Block. As default, it will be added at the end. Default: None.

   - **main_key**(string, optional) – Main keyword of the block that should be
     added (see: MKEYS) Default: the first main keyword of the file-type

   - **block**(keyword dict) – here the dict-keywords are the ogs-subkeywords and
     the value is the content that should be added with this ogs-subkeyword If a block
     should contain content directly connected to a main keyword, use this main keyword
     as input-keyword and the content as value: SUBKEY=content

add_content (self, content, main_index=None, sub_index=None, line_index=None)
   Add single-line content to the actual file.

Parameters

   - **content**(list) – list containing one line of content given as a list of single state-
     ments
• **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** (*int, optional*) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

• **line_index** (*int, optional*) – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

---

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

---

**add_copy_link** (*self, path, symlink=False*)

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

**Parameters**

• **path** (*str*) – path to the existing file that should be copied

• **symlink** (*bool, optional*) – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword** (*self, key, main_index=None*)

Add a new main keyword (#key) to the actual file.

**Parameters**

• **key** (*string*) – key name

• **main_index** (*int, optional*) – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

**add_multi_content** (*self, content, main_index=None, sub_index=None*)

Add multiple content to the actual file.

**Parameters**

• **content** (*list*) – list containing lines of content, each given as a list of single statements

• **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** (*int, optional*) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

---

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

---

**add_sub_keyword** (*self, key, main_index=None, sub_index=None*)

Add a new sub keyword ($key) to the actual file.

**Parameters**
• **key** *(string)* – key name

• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** *(int, optional)* – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

---

**Notes**

There needs to be at least one main keyword, otherwise the subkeyword is not added.

### append_to_block *(self, index=None, **block)*

Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

• **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

• ****block *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

### check *(self, verbose=True)*

Check if the given file is valid.

**Parameters**

• **verbose** *(bool, optional)* – Print information for the executed checks. Default: True

**Returns**

result – Validity of the given file.

**Return type**

bool

### del_block *(self, index=None, del_all=False)*

Delete a block by its index.

**Parameters**

• **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

• **del_all** *(bool, optional)* – State, if all blocks shall be deleted. Default: False

### del_content *(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)*

Delete content by its position.

**Parameters**

• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.

• **line_index** *(int, optional)* – position of the content line, that should be deleted. Default: -1
• **del_all** *(bool, optional)* – State, if all content shall be deleted. Default: False

**del_copy_link** *(self)*
Remove a former given link to an external file.

**del_main_keyword** *(self, main_index=None, del_all=False)*
Delete a main keyword (#key) by its position.

**Parameters**

• **main_index** *(int, optional)* – position, which main keyword should be deleted. Default: -1

• **del_all** *(bool, optional)* – State, if all main keywords shall be deleted. Default: False

**del_sub_keyword** *(self, main_index=-1, sub_index=-1, del_all=False)*
Delete a sub keyword ($key) by its position.

**Parameters**

• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• **pos** *(int, optional)* – position, which sub keyword should be deleted. Default: -1

• **del_all** *(bool, optional)* – State, if all sub keywords shall be deleted. Default: False

**get_block** *(self, index=None, as_dict=True)*
Get a Block from the actual file.

**Parameters**

• **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

• **as_dict** *(bool, optional)* – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for add_block. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no** *(self)*
Get the number of blocks in the file.

**get_file_type** *(self)*
Get the OGS file class name.

**get_multi_keys** *(self, index=None)*
State if a block has a unique set of sub keywords.

**is_block_unique** *(self, index=None)*
State if a block has a unique set of sub keywords.

**read_file** *(self, path, encoding=None, verbose=False)*
Read an existing OGS input file.

**Parameters**

• **path** *(str)* – path to the existing file that should be read

• **encoding** *(str or None, optional)* – encoding of the given file. If None is given, the system standard is used. Default: None

• **verbose** *(bool, optional)* – Print information of the reading process. Default: False

**reset** *(self)*
Delete every content.
save(self, path, **kwargs)
Save the actual OGS input file in the given path.

Parameters

• path (str) – path to where to file should be saved
• update (bool, optional) – state if the content should be updated before saving.
  Default: True

update_block(self, index=None, main_key=None, **block)
Update a Block from the actual file.

Parameters

• index (int or None, optional) – Positional index of the block of interest.
  As default, the last one is used. Default: None
• main_key (string, optional) – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None
• **block (keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

write_file(self)
Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['COMMUNICATION_TABLE']
SKEYS = [['MYRANK', 'NNEIGHBORS', 'NEIGHBOR']]
STD = {}
block_no
Number of blocks in the file.

file_name
base name of the file with extension.

    Type str

file_path
save path of the file.

    Type str

force_writing
state if the file is written even if empty.

    Type bool

is_empty
State if the OGS file is empty.

name
name of the file without extension.

    Type str
ogs5py.fileclasses.ddc

Class for the ogs DOMAIN DECOMPOSITION file.

File Class

```
DDC(**OGS_Config) Class for the ogs MPI DOMAIN DECOMPOSITION file.
```

```
class DDC(**OGS_Config)
    Bases: ogs5py.fileclasses.base.BlockFile
    Class for the ogs MPI DOMAIN DECOMPOSITION file.

Parameters

• task_root (str, optional) – Path to the destiny model folder. Default: cwd+"ogs5model"
• task_id (str, optional) – Name for the ogs task. Default: “model”
```

Notes

Main-Keywords (#):

• DOMAIN

Sub-Keywords ($) per Main-Keyword:

• DOMAIN
  – ELEMENTS
  – NODES_INNER
  – NODES_BORDER

Standard block: None

Keyword documentation: None

Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/par_ddc.cpp

See also:

```
add_block
```

Attributes

- `block_no` Number of blocks in the file.
- `file_name` str: base name of the file with extension.
- `file_path` str: save path of the file.
- `force_writing` bool: state if the file is written even if empty.
- `is_empty` State if the OGS file is empty.
- `name` str: name of the file without extension.

Methods
### 3.6. ogs5py.fileclasses

- **add_block** (self[, index, main_key])
  Add a new Block to the actual file.

- **add_content** (self, content[, main_index,...])
  Add single-line content to the actual file.

- **add_copy_link** (self, path[, symlink])
  Add a link to copy a file instead of writing.

- **add_main_keyword** (self, key[, main_index])
  Add a new main keyword (#key) to the actual file.

- **add_multi_content** (self, content[, ...])
  Add multiple content to the actual file.

- **add_sub_keyword** (self, key[, main_index, ...])
  Add a new sub keyword ($key) to the actual file.

- **append_to_block** (self[, index])
  Append data to an existing Block in the actual file.

- **check** (self[, verbose])
  Check if the given file is valid.

- **del_block** (self[, index, del_all])
  Delete a block by its index.

- **del_content** (self[, main_index, sub_index,...])
  Delete content by its position.

- **del_copy_link** (self)
  Remove a former given link to an external file.

- **del_main_keyword** (self[, main_index, del_all])
  Delete a main keyword (#key) by its position.

- **del_sub_keyword** (self[, main_index, del_all])
  Delete a sub keyword ($key) by its position.

- **get_block** (self[, index, as_dict])
  Get a Block from the actual file.

- **get_block_no** (self)
  Get the number of blocks in the file.

- **get_file_type** (self)
  Get the OGS file class name.

- **get_multi_keys** (self[, index])
  State if a block has a unique set of sub keywords.

- **is_block_unique** (self[, index])
  State if a block has a unique set of sub keywords.

- **read_file** (self, path[, encoding, verbose])
  Read an existing OGS input file.

- **reset** (self)
  Delete every content.

- **save** (self, path, **kwargs)
  Save the actual DDC input file in the given path.

- **update_block** (self[, index, main_key])
  Update a Block from the actual file.

- **write_file** (self)
  Write the actual OGS input file to the given folder.

---

**add_block** *(self, index=None, main_key=None, **block)*

Add a new Block to the actual file.

**Keywords**

- #MAIN_KEY
- **$SUBKEY1** content1 ...
- **$SUBKEY2** content2 ...

which looks like the following:

FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)

**Parameters**

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- **main_key** *(string, optional)* – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type
- ****block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**add_content** *(self, content, main_index=None, sub_index=None, line_index=None)*

Add single-line content to the actual file.

**Parameters**

- **content** *(list)* – list containing one line of content given as a list of single statements
main_index (int, optional) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

sub_index (int, optional) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

line_index(int, optional) – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

Notes
There needs to be at least one main keyword, otherwise the content is not added.
If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

add_copy_link (self, path, symlink=False)
Add a link to copy a file instead of writing.
Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

Parameters
• path (str) – path to the existing file that should be copied
• symlink (bool, optional) – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

add_main_keyword (self, key, main_index=None)
Add a new main keyword (#key) to the actual file.

Parameters
• key (string) – key name
• main_index (int, optional) – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

add_multi_content (self, content, main_index=None, sub_index=None)
Add multiple content to the actual file.

Parameters
• content (list) – list containing lines of content, each given as a list of single statements
• main_index (int, optional) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
• sub_index (int, optional) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

Notes
There needs to be at least one main keyword, otherwise the content is not added.
The content will be added at the end of the actual subkeyword.
If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

add_sub_keyword (self, key, main_index=None, sub_index=None)
Add a new sub keyword ($key) to the actual file.

Parameters
• **key** *(string)* – key name

• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** *(int, optional)* – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

---

**Notes**

There needs to be at least one main keyword, otherwise the subkeyword is not added.

---

```python
append_to_block(self, index=None, **block)
```

Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```python
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

- **block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: `SUBKEY=content`

```python
check(self, verbose=True)
```

Check if the given file is valid.

**Parameters**

- **verbose** *(bool, optional)* – Print information for the executed checks. Default: True

**Returns**

- **result** – Validity of the given file.

**Return type**

*bool*

```python
del_block(self, index=None, del_all=False)
```

Delete a block by its index.

**Parameters**

- **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

- **del_all** *(bool, optional)* – State, if all blocks shall be deleted. Default: False

```python
del_content(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)
```

Delete content by its position.

**Parameters**

- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.

- **line_index** *(int, optional)* – position of the content line, that should be deleted. Default: -1
• `del_all` *(bool, optional)* – State, if all content shall be deleted. Default: False

`del_copy_link` *(self)*
Remove a former given link to an external file.

`del_main_keyword` *(self, main_index=None, del_all=False)*
Delete a main keyword (#key) by its position.

**Parameters**

• `main_index` *(int, optional)* – position, which main keyword should be deleted. Default: -1

• `del_all` *(bool, optional)* – State, if all main keywords shall be deleted. Default: False

`del_sub_keyword` *(self, main_index=-1, sub_index=-1, del_all=False)*
Delete a sub keyword ($key) by its position.

**Parameters**

• `main_index` *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• `pos` *(int, optional)* – position, which sub keyword should be deleted. Default: -1

• `del_all` *(bool, optional)* – State, if all sub keywords shall be deleted. Default: False

`get_block` *(self, index=None, as_dict=True)*
Get a Block from the actual file.

**Parameters**

• `index` *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

• `as_dict` *(bool, optional)* – Here you can state if you want the output as a dictionary, which can be used as keyword-arguments for `add_block`. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

`get_block_no` *(self)*
Get the number of blocks in the file.

`get_file_type` *(self)*
Get the OGS file class name.

`get_multi_keys` *(self, index=None)*
State if a block has a unique set of sub keywords.

`is_block_unique` *(self, index=None)*
State if a block has a unique set of sub keywords.

`read_file` *(self, path, encoding=None, verbose=False)*
Read an existing OGS input file.

**Parameters**

• `path` *(str)* – path to the existing file that should be read

• `encoding` *(str or None, optional)* – encoding of the given file. If None is given, the system standard is used. Default: None

• `verbose` *(bool, optional)* – Print information of the reading process. Default: False

`reset` *(self)*
Delete every content.
**save** *(self, path, **kwargs)*  
Save the actual DDC input file in the given path.

**Parameters**

- **path** *(str)* – path to where to file should be saved
- **update** *(bool, optional)* – state if the content should be updated before saving. Default: True

**update_block** *(self, index=None, main_key=None, **block)*  
Update a Block from the actual file.

**Parameters**

- **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is used. Default: None
- **main_key** *(string, optional)* – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None
- ****block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**write_file** *(self)*  
Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

MKEYS = `["DOMAIN"]`

SKEYS = `[['ELEMENTS', 'NODES_INNER', 'NODES_BORDER']]`

STD = `{}`

**block_no**  
Number of blocks in the file.

**file_name**  
base name of the file with extension.

  Type **str**

**file_path**  
save path of the file.

  Type **str**

**force_writing**  
state if the file is written even if empty.

  Type **bool**

**is_empty**  
State if the OGS file is empty.

**name**  
name of the file without extension.

  Type **str**
ogs5py.fileclasses.fct

Class for the ogs FUNCTION file.

**File Class**

\[ \texttt{FCT(\*\*OGS\_Config)} \]

\textit{Class for the ogs FUNCTION file.}

\textbf{class FCT (\*\*OGS\_Config)}

\texttt{Bases: ogs5py.fileclasses.base.BlockFile}

\textit{Class for the ogs FUNCTION file.}

**Parameters**

- \texttt{task\_root (str, optional)} – Path to the destiny model folder. Default: cwd+"ogs5model"
- \texttt{task\_id (str, optional)} – Name for the ogs task. Default: “model”

**Notes**

\textbf{Main-Keywords (#):}

- FUNCTION

\textbf{Sub-Keywords ($) per Main-Keyword:}

- FUNCTION
  - DATA
  - DIMENSION
  - DIS\_TYPE
  - GEO\_TYPE
  - MATRIX
  - TYPE
  - VARIABLES

\textbf{Standard block: None}


\textbf{Reading routines:} https://github.com/ufz/ogs5/blob/master/FEM/rf_fct.cpp#L82

\textbf{See also:}

\texttt{add\_block}

\textbf{Attributes}

- \texttt{block\_no} \texttt{Number of blocks in the file.}
- \texttt{file\_name} \texttt{str: base name of the file with extension.}
- \texttt{file\_path} \texttt{str: save path of the file.}
- \texttt{force\_writing} \texttt{bool: state if the file is written even if empty.}
- \texttt{is\_empty} \texttt{State if the OGS file is empty.}
- \texttt{name} \texttt{str: name of the file without extension.}
### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_block</td>
<td>Add a new Block to the actual file.</td>
</tr>
<tr>
<td>add_content</td>
<td>Add single-line content to the actual file.</td>
</tr>
<tr>
<td>add_copy_link</td>
<td>Add a link to copy a file instead of writing.</td>
</tr>
<tr>
<td>add_main_keyword</td>
<td>Add a new main keyword (#key) to the actual file.</td>
</tr>
<tr>
<td>add_multi_content</td>
<td>Add multiple content to the actual file.</td>
</tr>
<tr>
<td>add_sub_keyword</td>
<td>Add a new sub keyword ($key) to the actual file.</td>
</tr>
<tr>
<td>append_to_block</td>
<td>Append data to an existing Block in the actual file.</td>
</tr>
<tr>
<td>check</td>
<td>Check if the given file is valid.</td>
</tr>
<tr>
<td>del_block</td>
<td>Delete a block by its index.</td>
</tr>
<tr>
<td>del_content</td>
<td>Delete content by its position.</td>
</tr>
<tr>
<td>del_copy_link</td>
<td>Remove a former given link to an external file.</td>
</tr>
<tr>
<td>del_main_keyword</td>
<td>Delete a main keyword (#key) by its position.</td>
</tr>
<tr>
<td>del_sub_keyword</td>
<td>Delete a sub keyword ($key) by its position.</td>
</tr>
<tr>
<td>get_block</td>
<td>Get a Block from the actual file.</td>
</tr>
<tr>
<td>get_file_type</td>
<td>Get the OGS file class name.</td>
</tr>
<tr>
<td>get_multi_keys</td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td>is_block_unique</td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td>read_file</td>
<td>Read an existing OGS input file.</td>
</tr>
<tr>
<td>reset</td>
<td>Delete every content.</td>
</tr>
<tr>
<td>save</td>
<td>Save the actual OGS input file in the given path.</td>
</tr>
<tr>
<td>update_block</td>
<td>Update a Block from the actual file.</td>
</tr>
<tr>
<td>write_file</td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>

**add_block**

```
add_block(self, index=None, main_key=None, **block)
```

Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY

$SUBKEY1  content1 ...

$SUBKEY2  content2 ...
```

which looks like the following:

```
FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- `index (int or None, optional)` – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- `main_key (string, optional)` – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type
- `**block (keyword dict)` – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**add_content**

```
add_content(self, content, main_index=None, sub_index=None, line_index=None)
```

Add single-line content to the actual file.

**Parameters**
• **content** ([list]) – list containing one line of content given as a list of single statements

• **main_index** ([int, optional]) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** ([int, optional]) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

• **line_index** ([int, optional]) – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

---

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

---

**add_copy_link** *(self, path, symlink=False)*

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

**Parameters**

- **path** (*str*) – path to the existing file that should be copied
- **symlink** (*bool, optional*) – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

---

**add_main_keyword** *(self, key, main_index=None)*

Add a new main keyword (#key) to the actual file.

**Parameters**

- **key** (*string*) – key name
- **main_index** ([int, optional]) – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

---

**add_multi_content** *(self, content, main_index=None, sub_index=None)*

Add multiple content to the actual file.

**Parameters**

- **content** ([list]) – list containing lines of content, each given as a list of single statements
- **main_index** ([int, optional]) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** ([int, optional]) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

---

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

---

**add_sub_keyword** *(self, key, main_index=None, sub_index=None)*

Add a new sub keyword ($key) to the actual file.
Parameters

• **key**(string) – key name

• **main_index**(int, optional) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index**(int, optional) – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

Notes

There needs to be at least one main keyword, otherwise the subkeyword is not added.

append_to_block (**self, index=None, **block**)

Append data to an existing Block in the actual file. Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```python
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

Parameters

• **index**(int or None, optional) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

• **block**(keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

check (**self, verbose=True**)

Check if the given file is valid.

Parameters verbose(bool, optional) – Print information for the executed checks. Default: True

Returns result – Validity of the given file.

Return type bool

del_block (**self, index=None, del_all=False**)

Delete a block by its index.

Parameters

• **index**(int or None, optional) – Positional index of the block of interest. As default, the last one is returned. Default: None

• del_all(bool, optional) – State, if all blocks shall be deleted. Default: False

del_content (**self, main_index=-1, sub_index=-1, line_index=-1, del_all=False**)

Delete content by its position.

Parameters

• main_index(int, optional) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• sub_index(int, optional) – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.
• **line_index** (*int*, *optional*) – position of the content line, that should be deleted. Default: -1

• **del_all** (*bool*, *optional*) – State, if all content shall be deleted. Default: False

**del_copy_link**(self)
Remove a former given link to an external file.

**del_main_keyword**(self, *main_index=None, del_all=False)
Delete a main keyword (#key) by its position.

Parameters

• **main_index** (*int*, *optional*) – position, which main keyword should be deleted. Default: -1

• **del_all** (*bool*, *optional*) – State, if all main keywords shall be deleted. Default: False

**del_sub_keyword**(self, *main_index=-1, sub_index=-1, del_all=False)
Delete a sub keyword ($key) by its position.

Parameters

• **main_index** (*int*, *optional*) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• **pos** (*int*, *optional*) – position, which sub keyword should be deleted. Default: -1

• **del_all** (*bool*, *optional*) – State, if all sub keywords shall be deleted. Default: False

**get_block**(self, *index=None, as_dict=True)
Get a Block from the actual file.

Parameters

• **index** (*int or None*, *optional*) – Positional index of the block of interest. As default, the last one is returned. Default: None

• **as_dict** (*bool*, *optional*) – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for add_block. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no**(self)
Get the number of blocks in the file.

**get_file_type**(self)
Get the OGS file class name.

**get_multi_keys**(self, *index=None*)
State if a block has a unique set of sub keywords.

**is_block_unique**(self, *index=None*)
State if a block has a unique set of sub keywords.

**read_file**(self, *path, encoding=None, verbose=False*)
Read an existing OGS input file.

Parameters

• **path** (*str*) – path to the existing file that should be read

• **encoding** (*str or None*, *optional*) – encoding of the given file. If None is given, the system standard is used. Default: None

• **verbose** (*bool*, *optional*) – Print information of the reading process. Default: False
reset(self)
    Delete every content.

save(self, path, **kwargs)
    Save the actual OGS input file in the given path.

    Parameters
    · path (str) – path to where to file should be saved
    · update (bool, optional) – state if the content should be updated before saving.
      Default: True

update_block(self, index=None, main_key=None, **block)
    Update a Block from the actual file.

    Parameters
    · index (int or None, optional) – Positional index of the block of interest.
      As default, the last one is used. Default: None
    · main_key (string, optional) – Main keyword of the block that should be updated (see: MKEYS)
      This shouldn’t be done. Default: None
    · **block (keyword dict) – here the dict-keywords are the ogs-subkeywords and
      the value is the content that should be added with this ogs-subkeyword
      If a block should contain content directly connected to a main keyword, use this main keyword
      as input-keyword and the content as value: SUBKEY=content

write_file(self)
    Write the actual OGS input file to the given folder.
    Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['FUNCTION']
SKEYS = [['TYPE', 'GEO_TYPE', 'DIS_TYPE', 'VARIABLES', 'DIMENSION', 'MATRIX', 'DATA']
STD = {}

block_no
    Number of blocks in the file.

file_name
    base name of the file with extension.
    Type str

file_path
    save path of the file.
    Type str

force_writing
    state if the file is written even if empty.
    Type bool

is_empty
    State if the OGS file is empty.

name
    name of the file without extension.
    Type str
ogs5py.fileclasses.gem

Class for the ogs GEOCHEMICAL THERMODYNAMIC MODELING COUPLING file.

File Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GEM</strong>(<strong>OGS_Config</strong>)</td>
<td>Class for the ogs GEOCHEMICAL THERMODYNAMIC MODELING COUPLING file.</td>
</tr>
<tr>
<td><strong>GEMinit</strong>([lst_name, dch, ipm, dbr, ...])</td>
<td>Class for GEMS3K input file.</td>
</tr>
</tbody>
</table>

**class GEM(****OGS_Config**)**

Bases: ogs5py.fileclasses.base.BlockFile

Class for the ogs GEOCHEMICAL THERMODYNAMIC MODELING COUPLING file.

**Parameters**

- **task_root** *(str, optional)* – Path to the destiny model folder. Default: cwd+"ogs5model"
- **task_id** *(str, optional)* – Name for the ogs task. Default: “model”

**Notes**

Main-Keywords (#):

- GEM_PROPERTIES

Sub-Keywords ($) per Main-Keyword:

- GEM_PROPERTIES
  - CALCULATE_BOUNDARY_NODES
  - DISABLE_GEMS
  - FLAG_COUPLING_HYDROLOGY
  - FLAG_DISABLE_GEM
  - FLAG_POROSITY_CHANGE
  - GEM_CALCULATE_BOUNDARY_NODES
  - GEM_INIT_FILE
  - GEM_THREADS
  - ITERATIVE_SCHEME
  - KINETIC_GEM
  - MAX_FAILED_NODES
  - MAX_POROSITY
  - MIN_POROSITY
  - MY_SMART_GEMS
  - PRESSURE_GEM
  - TEMPERATURE_GEM
  - TRANSPORT_B

**Standard block:** None
Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rfc_REACT_GEM.cpp#L2644

See also:

add_block

Attributes

block_no Number of blocks in the file.
file_name str: base name of the file with extension.
file_path str: save path of the file.
force_writing bool: state if the file is written even if empty.
is_empty State if the OGS file is empty.
name str: name of the file without extension.

Methods

add_block(self[, index, main_key]) Add a new Block to the actual file.
add_content(self[, content[, main_index, ...]]) Add single-line content to the actual file.
add_copy_link(self[, path[, symlink]]) Add a link to copy a file instead of writing.
add_main_keyword(self[, key[, main_index]]) Add a new main keyword (#key) to the actual file.
add_multi_content(self[, content[, ...]]) Add multiple content to the actual file.
add_sub_keyword(self[, key[, main_index, ...]]) Add a new sub keyword ($key) to the actual file.
append_to_block(self[, index]) Append data to an existing Block in the actual file.
check(self[, verbose]) Check if the given file is valid.
del_block(self[, index, del_all]) Delete a block by its index.
del_content(self[, main_index, sub_index, ...]) Delete content by its position.
del_copy_link(self) Remove a former given link to an external file.
del_main_keyword(self[, main_index, del_all]) Delete a main keyword (#key) by its position.
del_sub_keyword(self[, main_index, ...]) Delete a sub keyword ($key) by its position.
get_block(self[, index[, as_dict]]) Get a Block from the actual file.
get_block_no(self) Get the number of blocks in the file.
get_file_type(self) Get the OGS file class name.
get_multi_keys(self[, index]) State if a block has a unique set of sub keywords.
is_block_unique(self[, index]) State if a block has a unique set of sub keywords.
read_file(self[, path[, encoding[, verbose]]]) Read an existing OGS input file.
reset(self) Delete every content.
save(self[, path, \*\*kwargs]) Save the actual OGS input file in the given path.
update_block(self[, index, main_key]) Update a Block from the actual file.
write_file(self) Write the actual OGS input file to the given folder.

add_block (self, index=None, main_key=None, **block)
Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
which looks like the following:

```python
FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- `index (int or None, optional)` – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- `main_key (string, optional)` – Main keyword of the block that should be added (see: MKEYS). Default: the first main keyword of the file-type
- `**block (keyword dict)` – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**add_content** *(self, content, main_index=None, sub_index=None, line_index=None)*

Add single-line content to the actual file.

**Parameters**

- `content (list)` – list containing one line of content given as a list of single statements
- `main_index (int, optional)` – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- `sub_index (int, optional)` – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- `line_index (int, optional)` – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

**add_copy_link** *(self, path, symlink=False)*

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

**Parameters**

- `path (str)` – path to the existing file that should be copied
- `symlink (bool, optional)` – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword** *(self, key, main_index=None)*

Add a new main keyword (#key) to the actual file.

**Parameters**

- `key (string)` – key name
- `main_index (int, optional)` – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

**add_multi_content** *(self, content, main_index=None, sub_index=None)*

Add multiple content to the actual file.

**Parameters**
• **content** (*list*) – list containing lines of content, each given as a list of single statements

• **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** (*int, optional*) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

Notes
There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

**add_sub_keyword** *(self, key, main_index=None, sub_index=None)*
Add a new sub keyword ($key) to the actual file.

Parameters

• **key** (*string*) – key name

• **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** (*int, optional*) – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

Notes
There needs to be at least one main keyword, otherwise the subkeyword is not added.

**append_to_block** *(self, index=None, **block)*
Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```plaintext
#MAIN_KEY
$SUBKEY1 content1 . . .
$SUBKEY2 content2 . . .
```

which looks like the following:

FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)

Parameters

• **index** (*int or None, optional*) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

• **block** (*keyword dict*) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**check** *(self, verbose=True)*
Check if the given file is valid.

Parameters **verbose** (*bool, optional*) – Print information for the executed checks.

Default: True

Returns **result** – Validity of the given file.
Return type  bool

del_block (self, index=None, del_all=False)
Delete a block by its index.

Parameters

- **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is returned. Default: None
- **del_all** (*bool, optional*) – State, if all blocks shall be deleted. Default: False

del_content (self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)
Delete content by its position.

Parameters

- **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- **sub_index** (*int, optional*) – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.
- **line_index** (*int, optional*) – position of the content line, that should be deleted. Default: -1
- **del_all** (*bool, optional*) – State, if all content shall be deleted. Default: False

del_copy_link (self)
Remove a former given link to an external file.

del_main_keyword (self, main_index=None, del_all=False)
Delete a main keyword (#key) by its position.

Parameters

- **main_index** (*int, optional*) – position, which main keyword should be deleted. Default: -1
- **del_all** (*bool, optional*) – State, if all main keywords shall be deleted. Default: False

del_sub_keyword (self, main_index=-1, sub_index=-1, del_all=False)
Delete a sub keyword ($key) by its position.

Parameters

- **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- **pos** (*int, optional*) – position, which sub keyword should be deleted. Default: -1
- **del_all** (*bool, optional*) – State, if all sub keywords shall be deleted. Default: False

get_block (self, index=None, as_dict=True)
Get a Block from the actual file.

Parameters

- **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is returned. Default: None
- **as_dict** (*bool, optional*) – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for add_block. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

get_block_no (self)
Get the number of blocks in the file.
get_file_type (self)
Get the OGS file class name.

get_multi_keys (self, index=None)
State if a block has a unique set of sub keywords.

is_block_unique (self, index=None)
State if a block has a unique set of sub keywords.

read_file (self, path, encoding=None, verbose=False)
Read an existing OGS input file.

Parameters

• path (str) – path to the existing file that should be read
• encoding (str or None, optional) – encoding of the given file. If None is given, the system standard is used. Default: None
• verbose (bool, optional) – Print information of the reading process. Default: False

reset (self)
Delete every content.

save (self, path, **kwargs)
Save the actual OGS input file in the given path.

Parameters

• path (str) – path to where to file should be saved
• update (bool, optional) – state if the content should be updated before saving. Default: True

update_block (self, index=None, main_key=None, **block)
Update a Block from the actual file.

Parameters

• index (int or None, optional) – Positional index of the block of interest. As default, the last one is used. Default: None
• main_key (string, optional) – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None
• **block (keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

write_file (self)
Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['GEM_PROPERTIES']
SKEYS = [['GEM_INIT_FILE', 'GEM_THREADS', 'TRANSPORT_B', 'FLAG_POROSITY_CHANGE', 'MIN_POROSITY', 'MAX_POROSITY', ...
STD = {}

block_no
Number of blocks in the file.

file_name
base name of the file with extension.

Type str
file_path
    save path of the file.
    Type str

force_writing
    state if the file is written even if empty.
    Type bool

is_empty
    State if the OGS file is empty.

name
    name of the file without extension.
    Type str

class GEMinit(lst_name='model-dat.lst', dch=None, ipm=None, dbr=None, task_root=None, task_id='model')
    Bases: object
    Class for GEMS3K input file.
    lst file that contains the names of
    • the GEMS data file (dch file),
    • the GEMS numerical settings (ipm file)
    • the example setup (dbr file)
    used to initialize the GEMS3K kernel.

    Parameters
    • lst_name (str or None, optional) – name of the lst file
    • dch (LineFile or None) – the GEMS data file
    • ipm (LineFile or None) – the GEMS data file
    • dbr (LineFile or None) – the GEMS data file
    • task_root (str, optional) – Path to the destiny model folder. Default: cwd+"ogs5model"
    • task_id (str, optional) – Name for the ogs task. Default: “model”

Notes
http://gems.web.psi.ch/GEMS3K/

Attributes
    file_ext  The extension of the lst file.
    file_names The names of the included files.
    files List of the included files: dch, ipm, dbr.
    is_empty  State if the file is empty.
    name The name of the lst file.
    task_root Get and set the task_root path of the ogs model.
Methods

```
check(self[, verbose])
Check if the GEM external file is valid.

get_file_type(self)
Get the OGS file class name.

read_file(self, path[, encoding, verbose])
Read a given GEM external input lst-file.

reset(self)
Delete every content.

save(self, path)
Save the actual GEM external file in the given path.

write_file(self)
Write the actual OGS input file to the given folder.
```

```
check (self, verbose=True)
Check if the GEM external file is valid.

    Parameters verbose (bool, optional) – Print information for the executed checks.
    Default: True

    Returns result – Validity.

    Return type  bool

get_file_type (self)
Get the OGS file class name.

read_file (self, path, encoding=None, verbose=False)
Read a given GEM external input lst-file.

    Parameters path (str) – path to the file

Notes
This also reads the given files in the lst-file. (dch, ipm, dbr)
```

```
reset (self)
Delete every content.

save (self, path)
Save the actual GEM external file in the given path.

    1st file containing: dch, ipm, dbr

    Parameters path (str) – path to where to file should be saved

write_file (self)
Write the actual OGS input file to the given folder.

    Its path is given by “task_root+task_id+file_ext”.

file_ext
The extension of the lst file.

file_names
The names of the included files.

files
dch, ipm, dbr.

    Type  List of the included files

is_empty
State if the file is empty.

name
The name of the lst file.

task_root
Get and set the task_root path of the ogs model.
```
ogs5py Documentation, Release 1.1.1

ogs5py.fileclasses.gli

Class for the ogs GEOMETRY file.

Subpackages

The generators can be called with `GLI.generate`

```
generator
```
Generators for the ogs GEOMETRY file.

File Classes

```
GLI([gli_dict])

GLIext([typ, data, name, file_ext, ...])
```
Class for the ogs GEOMETRY file.

Class for an external definition for the ogs GEOMETRY file.

class GLI (gli_dict=None, **OGS_Config)

Bases: ogs5py.fileclasses.base.File

Class for the ogs GEOMETRY file.

Parameters

- **gli_dict** (dict or None, optional) – dictionary containing the gli file Includes the following information (sorted by keys):
  - **points** [ndarray] Array with all point postions
  - **point_names** [ndarray (of strings)] Array with all point names
  - **point_md** [ndarray] Array with all Material-densities at the points if point_md should be undefined it takes the value -np.inf
  - **polylinies** [list of dict] each containing information about
    - **ID** (int or None)
    - **NAME** (str)
    - **POINTS** (ndarray)
    - **EPSILON** (float or None)
    - **TYPE** (int or None)
    - **MAT_GROUP** (int or None)
    - **POINT_VECTOR** (str or None)
  - **surfaces** [list of dict] each containing information about
    - **ID** (int or None)
    - **NAME** (str)
    - **POLYLINES** (list of str)
    - **EPSILON** (float or None)
    - **TYPE** (int or None)
    - **MAT_GROUP** (int or None)
    - **TIN** (str or None)
volumes [list of dict] each containing information about
- NAME (str)
- SURFACES (list of str)
- TYPE (int or None)
- MAT_GROUP (int or None)
- LAYER (int or None)

Default: None

• **task_root** (str, optional) – Path to the destiny model folder. Default: cwd+“ogs5model”

• **task_id** (str, optional) – Name for the ogs task. Default: “model”

## Attributes

- **POINTS** ndarray: POINTS (n,3) of the gli, defined by xyz-coordinates.
- **POINT_MD** ndarray: material density values of POINTS of the gli.
- **POINT_NAMES** ndarray: names of POINTS of the gli.
- **POINT_NO** int: number of POINTS of the gli.
- **POLYLINES** List of dict: POLYLINES of the gli.
- **POLYLINE_NAMES** List of str: names of POLYLINES of the gli.
- **POLYLINE_NO** int: number of POLYLINES of the gli.
- **SURFACES** List of dict: SURFACES of the gli.
- **SURFACE_NAMES** List of str: names of SURFACES of the gli.
- **SURFACE_NO** int: number of SURFACES of the gli.
- **VOLUMES** List of dict: VOLUMES of the gli.
- **VOLUME_NAMES** List of str: names of VOLUMES of the gli.
- **VOLUME_NO** int: number of VOLUMES of the gli.
- **file_name** str: base name of the file with extension.
- **file_path** str: save path of the file.
- **force_writing** bool: state if the file is written even if empty.
- **is_empty** bool: State if the GLI File is empty.
- **name** str: name of the file without extension.

## Methods

- **__call__(self)**: Return a copy of the underlying dictionary of the gli.
- **add_copy_link(self, path[, symlink])**: Add a link to copy a file instead of writing.
- **add_points(self, points[, names, decimals])**: Add a list of points (ndarray with shape (n,3)).
- **add_polyline(self, name, points[, ply_id, ...])**: Add a polyline to the gli.
- **add_surface(self, name, polylines[, srf_id, ...])**: Add a new surface.

Continued on next page
### Table 34 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_volume(self, name, surfaces[, vol_type, ...])</code></td>
<td>Add a new volume.</td>
</tr>
<tr>
<td><code>check(self[, verbose])</code></td>
<td>Check if the gli is valid.</td>
</tr>
<tr>
<td><code>delete_copy_link(self)</code></td>
<td>Remove a former given link to an external file.</td>
</tr>
<tr>
<td><code>generate(self[, generator])</code></td>
<td>Use a gli-generator from the generator module.</td>
</tr>
<tr>
<td><code>get_file_type(self)</code></td>
<td>Get the OGS file class name.</td>
</tr>
<tr>
<td><code>load(self, filepath[, verbose, encoding])</code></td>
<td>Load an OGS5 gli from file.</td>
</tr>
<tr>
<td><code>pnt_coord(self[, pnt_name, pnt_id])</code></td>
<td>Get Point coordinates either by name or ID.</td>
</tr>
<tr>
<td><code>read_file(self[, path, encoding, verbose])</code></td>
<td>Load an OGS5 gli from file.</td>
</tr>
<tr>
<td><code>remove_point(self, id_or_name)</code></td>
<td>Remove a point by its name or ID.</td>
</tr>
<tr>
<td><code>remove_polyline(self, names)</code></td>
<td>Remove a polyline by its name.</td>
</tr>
<tr>
<td><code>remove_surface(self, names)</code></td>
<td>Remove a surface by its name.</td>
</tr>
<tr>
<td><code>remove_volume(self, names)</code></td>
<td>Remove a volume by its name.</td>
</tr>
<tr>
<td><code>reset(self)</code></td>
<td>Delete every content.</td>
</tr>
<tr>
<td><code>rotate(self, angle[, rotation_axis, ...])</code></td>
<td>Rotate points around a given rotation point and axis with given angle.</td>
</tr>
</tbody>
</table>
| `save(self, path, \*
\*
\*kwkwargs)`       | Save the gli to an OGS5 gli file.                                          |
| `set_dict(self, gli_dict)`                  | Set a gli dict as returned by tools methods or generators.                |
| `shift(self, vector)`                       | Shift points with a given vector.                                          |
| `swap_axis(self[, axis1, axis2])`           | Swap axis of the coordinate system.                                       |
| `write_file(self)`                          | Write the actual OGS input file to the given folder.                       |

### __call__ (self)

Return a copy of the underlying dictionary of the gli.

**Returns** Mesh – dictionary representation of the mesh

**Return type** dict

### `add_copy_link` (self, path, symlink=False)

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linkied to the target folder.

**Parameters**

- **path** *(str)* – path to the existing file that should be copied
- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

### `add_points` (self, points, names=None, md=None, decimals=4)

Add a list of points (ndarray with shape (n,3)).

Keeps the pointlist unique. If a named point is added, that was already present, it will be renamed with the new name. Same for md. The pointlists of the polylines will be updated.

**Parameters**

- **points** *(ndarray)* – Array with new points.
- **names** *(ndarray of str or None, optional)* – array containing the names. If None, all new points are unnamed. Default: None
- **md** *(ndarray of float or None, optional)* – array containing the material density. If None, all new points will have unspecified md. Default: None
- **decimals** *(int, optional)* – Number of decimal places to round the added points to (default: 4). If decimals is negative, it specifies the number of positions...
to the left of the decimal point. This will not round the new points, it’s just for comparison of the already present points to guarantee uniqueness.

**Returns** `new_pos` – array with the IDs of the added points in the pointlist of the gli.

**Return type** `ndarray`

**add_polyline**

```python
add_polyline(self, name, points, ply_id=None, epsilon=None, ply_type=None, mat_group=None, point_vector=None, closed=False, decimals=4)
```

Add a polyline to the gli.

**Parameters**

- `points (ndarray)` – Array with new points. Either of shape (n,3) to add new points by their coordinates or a list of points IDs referring to existing points.
- `name (str)` – name of the new polyline
- `points` – Array with the points. Either array of point IDs or new coordinates.
- `ply_id (int or None, optional)` – Default: None
- `epsilon (float or None, optional)` – Default: None
- `ply_type (int or None, optional)` – Default: None
- `mat_group (int or None, optional)` – Default: None
- `point_vector (str or None, optional)` – Default: None
- `closed (bool, optional)` – If the polyline shall be closed, the first point will be added as last point again. Default: False
- `decimals (int, optional)` – Number of decimal places to round the added points to (default: 4). If decimals is negative, it specifies the number of positions to the left of the decimal point. This will not round the new points, it’s just for comparison of the already present points to guarantee uniqueness.

**add_surface**

```python
add_surface(self, name, polylines, srf_id=None, epsilon=None, srf_type=0, mat_group=None, tin=None)
```

Add a new surface.

**Parameters**

- `name (str)` – name of the new surface
- `polylines (list of str)` – List of the surface-defining polyline-names
- `srf_id (int or None, optional)` – Default: None
- `epsilon (float or None, optional)` – Default: None
- `srf_type (int or None, optional)` – Default: None
- `mat_group (int or None, optional)` – Default: None
- `tin (str or None, optional)` – Default: None

**add_volume**

```python
add_volume(self, name, surfaces, vol_type=None, mat_group=None, layer=None)
```

Add a new volume.

**Parameters**

- `name (str)` – name of the new surface
- `surfaces (list of str)` – List of the volume-defining surface-names
- `vol_type (int or None, optional)` – Default: None
- `mat_group (int or None, optional)` – Default: None
- `layer (int or None, optional)` – Default: None
check (self, verbose=True)
    Check if the gli is valid.

    In the sense, that the contained data is consistent.

    Parameters verbose (bool, optional) – Print information for the executed
    checks. Default: True

    Returns result – Validity of the given gli.

    Return type bool

del_copy_link (self)
    Remove a former given link to an external file.

generate (self, generator='rectangular', **kwargs)
    Use a gli-generator from the generator module.

    See: ogs5py.fileclasses.gli.generator

    Parameters

    • generator (str) – set the generator from the generator module

    • **kwargs – kwargs will be forwarded to the generator in use

    Notes

    The following generators are available:

    | generator     | Description                              |
    |---------------|------------------------------------------|
    | rectangular   | Generate a rectangular boundary for a grid in 2D or 3D as gli. |
    | radial        | Generate a radial boundary for a grid in 2D or 3D. |

get_file_type (self)
    Get the OGS file class name.

load (self, filepath, verbose=False, encoding=None, **kwargs)
    Load an OGS5 gli from file.

    kwargs will be forwarded to “tools.load_ogs5gli”

    Parameters

    • filepath (string) – path to the ‘*.gli’ OGS5 gli file to load

    • verbose (bool, optional) – Print information of the reading process. De-
      fault: True

pnt_coord (self, pnt_name=None, pnt_id=None)
    Get Point coordinates either by name or ID.

    Parameters

    • pnt_name (str) – Point name.

    • pnt_id (int) – Point ID.

read_file (self, path, encoding=None, verbose=False)
    Load an OGS5 gli from file.

    Parameters

    • path (string) – path to the ‘*.gli’ OGS5 gli file to load

    • encoding (str or None, optional) – encoding of the given file. If
      None is given, the system standard is used. Default: None
• **verbose** (*bool, optional*) – Print information of the reading process. Default: False

**remove_point** (*self, id_or_name*)
Remove a point by its name or ID.

If Points are removed, that define polylines, they will be removed. Same for surfaces and volumes.

**Parameters**

- **id_or_name** (*int or str or list of int or list of str*) – Points to be removed. Unknown names or IDs are ignored.

**remove_polyline** (*self, names*)
Remove a polyline by its name.

If Polylines are removed, that define surfaces, they will be removed. Same for volumes.

**Parameters**

- **names** (*str or list of str*) – Polylines to be removed. Unknown names are ignored.

**remove_surface** (*self, names*)
Remove a surface by its name.

If Surfaces are removed, that define Volumes, they will be removed.

**Parameters**

- **names** (*str or list of str*) – Surfaces to be removed. Unknown names are ignored.

**remove_volume** (*self, names*)
Remove a volume by its name.

**Parameters**

- **names** (*str or list of str*) – Volumes to be removed. Unknown names are ignored.

**reset** (*self*)
Delete every content.

**rotate** (*self, angle, rotation_axis=(0.0, 0.0, 1.0), rotation_point=(0.0, 0.0, 0.0))
Rotate points around a given rotation point and axis with given angle.

**Parameters**

- **angle** (*float*) – rotation angle given in radial length
- **rotation_axis** (*array_like, optional*) – Array containing the vector for rotation axis. Default: (0,0,1)
- **rotation_point** (*array_like, optional*) – Vector of the rotation base point. Default:(0,0,0)

**save** (*self, path, **kwargs*)
Save the gli to an OGS5 gli file.

**Parameters**

- **path** (*string*) – path to the '*.gli' OGS5 gli file to save
- **verbose** (*bool, optional*) – Print information of the writing process. Default: True

**set_dict** (*self, gli_dict*)
Set a gli dict as returned by tools methods or generators.

Gli will be checked for validity.

**Parameters**

- **gli_dict** (*dict*) – dictionary containing the gli file Includes the following information (sorted by keys):
  - **points** [ndarray] Array with all point postions
  - **point_names** [ndarray (of strings)] Array with all point names
**point_md** [ndarray] Array with all Material-densities at the points if point_md should be undefined it takes the value -np.inf

**polylines** [list of dict] each containing information about

- **ID** (int or None)
- **NAME** (str)
- **POINTS** (ndarray)
- **EPSILON** (float or None)
- **TYPE** (int or None)
- **MAT_GROUP** (int or None)
- **POINT_VECTOR** (str or None)

**surfaces** [list of dict] each containing information about

- **ID** (int or None)
- **NAME** (str)
- **POLYLINES** (list of str)
- **EPSILON** (float or None)
- **TYPE** (int or None)
- **MAT_GROUP** (int or None)
- **TIN** (str or None)

**volumes** [list of dict] each containing information about

- **NAME** (str)
- **SURFACES** (list of str)
- **TYPE** (int or None)
- **MAT_GROUP** (int or None)
- **LAYER** (int or None)

**shift** *(self, vector)*

Shift points with a given vector.

**Parameters**

- **vector** *(ndarray)* – array containing the shifting vector

**swap_axis** *(self, axis1='y', axis2='z')*

Swap axis of the coordinate system.

**Parameters**

- **axis1** *(str or int, optional)* – First selected Axis. Either in [“x”, “y”, “z”] or in [0, 1, 2]. Default: “y”
- **axis2** *(str or int, optional)* – Second selected Axis. Either in [“x”, “y”, “z”] or in [0, 1, 2]. Default: “z”

**write_file** *(self)*

Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

**POINTS**

POINTS (n,3) of the gli, defined by xyz-coordinates.

**Type** ndarray

**POINT_MD**

material density values of POINTS of the gli.
Type ndarray

**POINT_NAMES**
names of POINTS of the gli.

Type ndarray

**POINT_NO**
number of POINTS of the gli.

Type int

**POLYLINES**
POLYLINES of the gli.

Type List of dict

**POLYLINE_NAMES**
names of POLYLINES of the gli.

Type List of str

**POLYLINE_NO**
number of POLYLINES of the gli.

Type int

**SURFACES**
SURFACES of the gli.

Type List of dict

**SURFACE_NAMES**
names of SURFACES of the gli.

Type List of str

**SURFACE_NO**
number of SURFACES of the gli.

Type int

**VOLUMES**
VOLUMES of the gli.

Type List of dict

**VOLUME_NAMES**
names of VOLUMES of the gli.

Type List of str

**VOLUME_NO**
number of VOLUMES of the gli.

Type int

**file_name**
base name of the file with extension.

Type str

**file_path**
save path of the file.

Type str

**force_writing**
state if the file is written even if empty.

Type bool
is_empty
State if the GLI File is empty.

Type bool

name
name of the file without extension.

Type str
class GLIext(typ='TIN', data=None, name=None, file_ext=None, task_root=None, task_id='model')
Bases: ogs5py.fileclasses.base.File

Class for an external definition for the ogs GEOMETRY file.

Parameters

- typ (str, optional) – Type of the external geometry definition. Either TIN for a
  triangulated surface or POINT_VECTOR for a polyline. Default: "TIN"
- data (numpy.ndarray, optional) – Data for the external geometry definition.
  Default: None
- name (str, optional) – File name for the RFR file. If None, the task_id is
  used. Default: None
- file_ext (str, optional) – extension of the file (with leading dot ".rfr") Default:
  ".rfr"
- task_root (str, optional) – Path to the destiny model folder. Default:
cwd+"ogs5model"
- task_id (str, optional) – Name for the ogs task. Default: “model”

Attributes

- file_name str: base name of the file with extension.
- file_path str: save path of the file.
- force_writing bool: state if the file is written even if empty.
- is_empty State if the OGS file is empty.
- name str: name of the file without extension.

Methods

+ add_copy_link(self, path[, symlink])
  Add a link to copy a file instead of writing.
  Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target
  folder.

  Parameters

  - path (str) – path to the existing file that should be copied
• **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**check**(self, verbose=True)
Check if the external geometry definition is valid.

In the sense, that the contained data is consistent.

**Parameters verbose** *(bool, optional)* – Print information for the executed checks. Default: True

**Returns result** – Validity of the given gli.

**Return type** `bool`

**del_copy_link**(self)
Remove a former given link to an external file.

**get_file_type**(self)
Get the OGS file class name.

**read_file**(self, path, **kwargs)
Read a given GLI_EXT input file.

**Parameters path** *(str)* – path to the file

**reset**(self)
Delete every content.

**save**(self, path)
Save the actual GLI external file in the given path.

**Parameters path** *(str)* – path to where to file should be saved

**write_file**(self)
Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

**file_name**
base name of the file with extension.

**Type** `str`

**file_path**
save path of the file.

**Type** `str`

**force_writing**
state if the file is written even if empty.

**Type** `bool`

**is_empty**
State if the OGS file is empty.

**name**
name of the file without extension.

**Type** `str`
ogs5py.fileclasses.gli.generator

Generators for the ogs GEOMETRY file.

Generators

These generators can be called with `GLI.generate`

<table>
<thead>
<tr>
<th>generator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rectangular</td>
<td>Generate a rectangular boundary for a grid in 2D or 3D as gli.</td>
</tr>
<tr>
<td>radial</td>
<td>Generate a radial boundary for a grid in 2D or 3D.</td>
</tr>
</tbody>
</table>

**radial**

```python
dim=3, ori=(0.0, 0.0, 0.0), angles=16, rad_out=10.0, rad_in=None, z_size=-1.0,
name_out='boundary', name_in='well'
```

Generate a radial boundary for a grid in 2D or 3D.

**Parameters**

- `dim (int, optional)` – Dimension of the resulting mesh, either 2 or 3. Default: 3
- `ori (list of float, optional)` – Origin of the mesh. Default: [0.0, 0.0, 0.0]
- `angles (int, optional)` – Number of angles. Default: 16
- `rad_out (float or None, optional)` – Radius of the outer boundary. Default: 10.
- `rad_in` – Radius of the inner boundary if needed. (i.e. the well)
- `z_size (float, optional)` – Size of the mesh in z-direction
- `name_out (str, optional)` – Name of the outer boundary. In 3D there will be as many surfaces as angles are given. Their names are generated by adding the angle number: “_0”, “_1”, … Default: “boundary”
- `name_in (str, optional)` – Name of the inner boundary. In 3D there will be as many surfaces as angles are given. Their names are generated by adding the angle number: “_0”, “_1”, … Default: “well”

**Returns**

- `result` gli

**rectangular**

```python
dim=2, ori=(0.0, 0.0, 0.0), size=(10.0, 10.0, 10.0), name='boundary'
```

Generate a rectangular boundary for a grid in 2D or 3D as gli.

**Parameters**

- `dim (int, optional)` – Dimension of the resulting mesh, either 2 or 3. Default: 3
- `ori (list of float, optional)` – Origin of the mesh. Default: [0.0, 0.0, 0.0]
- `size (list of float, optional)` – Size of the mesh. Default: [10.0, 10.0, 10.0]
- `name (str, optional)` – Name of the boundary. In 3D there will be 4 surfaces where the names are generated by adding an ID: “_0”, “_1”, “_2”, “_3” Default: “boundary”

**Returns**

- `result`
Return type gli
ogs5py.fileclasses.ic

Class for the ogs INITIAL_CONDITION file.

File Classes

<table>
<thead>
<tr>
<th>IC(**OGS_Config)</th>
<th>Class for the ogs INITIAL_CONDITION file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFR([variables, data, units, headers, name, ...])</td>
<td>Class for the ogs RESTART file, if the DIS_TYPE in IC is set to RESTART.</td>
</tr>
</tbody>
</table>

```python
class IC(**OGS_Config)
    Bases: ogs5py.fileclasses.base.BlockFile

    Class for the ogs INITIAL_CONDITION file.

    Parameters

    - `task_root (str, optional)` – Path to the destiny model folder. Default: cwd+"ogs5model"
    - `task_id (str, optional)` – Name for the ogs task. Default: “model”

Notes

Main-Keywords (#):
- INITIAL_CONDITION

Sub-Keywords ($) per Main-Keyword:
- INITIAL_CONDITION
  - PCS_TYPE
  - PRIMARY_VARIABLE
  - COMP_NAME
  - STORE_VALUES
  - DIS_TYPE
  - GEO_TYPE

Standard block:
- PCS_TYPE  “GROUNDWATER_FLOW”
- PRIMARY_VARIABLE  “HEAD”
- GEO_TYPE  “DOMAIN”
- DIS_TYPE  [“CONSTANT”, 0.0]


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rf_ic_new.cpp#L222

See also:
- add_block

Attributes

- `block_no` Number of blocks in the file.
```
**file_name**  *str*: base name of the file with extension.

**file_path**  *str*: save path of the file.

**force_writing**  *bool*: state if the file is written even if empty.

**is_empty**  State if the OGS file is empty.

**name**  *str*: name of the file without extension.

### Methods

- **add_block**(self[, index, main_key])  
  Add a new Block to the actual file.

- **add_content**(self, content[, main_index, ...])  
  Add single-line content to the actual file.

- **add_copy_link**(self, path[, symlink])  
  Add a link to copy a file instead of writing.

- **add_main_keyword**(self, key[, main_index])  
  Add a new main keyword (#key) to the actual file.

- **add_multi_content**(self, content[, ...])  
  Add multiple content to the actual file.

- **add_sub_keyword**(self[, key[, main_index, ...]])  
  Add a new sub keyword ($key) to the actual file.

- **append_to_block**(self[, index])  
  Append data to an existing Block in the actual file.

- **check**(self[, verbose])  
  Check if the given file is valid.

- **del_block**(self[, index, del_all])  
  Delete a block by its index.

- **del_content**(self[, main_index, sub_index, ...])  
  Delete content by its position.

- **del_copy_link**(self)  
  Remove a former given link to an external file.

- **del_main_keyword**(self[, main_index, del_all])  
  Delete a main keyword (#key) by its position.

- **del_sub_keyword**(self[, main_index, ...])  
  Delete a sub keyword ($key) by its position.

- **get_block**(self[, index, as_dict])  
  Get a Block from the actual file.

- **get_block_no**(self)  
  Get the number of blocks in the file.

- **get_file_type**(self)  
  Get the OGS file class name.

- **get_multi_keys**(self[, index])  
  State if a block has a unique set of sub keywords.

- **is_block_unique**(self[, index])  
  State if a block has a unique set of sub keywords.

- **read_file**(self, path[, encoding, verbose])  
  Read an existing OGS input file.

- **reset**(self)  
  Delete every content.

- **save**(self, path, **kwargs)  
  Save the actual OGS input file in the given path.

- **update_block**(self[, index, main_key])  
  Update a Block from the actual file.

- **write_file**(self)  
  Write the actual OGS input file to the given folder.

### add_block

Add a new Block to the actual file.

**Keywords are the sub keywords of the actual file type:**

```plaintext
#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```plaintext
FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

- **main_key** *(string, optional)* – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type
**block** (keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

```python
add_content(self, content, main_index=None, sub_index=None, line_index=None)
```

Add single-line content to the actual file.

**Parameters**

- **content** *(list)* – list containing one line of content given as a list of single statements
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- **line_index** *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

```python
add_copy_link(self, path, symlink=False)
```

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

**Parameters**

- **path** *(str)* – path to the existing file that should be copied
- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

```python
add_main_keyword(self, key, main_index=None)
```

Add a new main keyword (#key) to the actual file.

**Parameters**

- **key** *(string)* – key name
- **main_index** *(int, optional)* – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

```python
add_multi_content(self, content, main_index=None, sub_index=None)
```

Add multiple content to the actual file.

**Parameters**

- **content** *(list)* – list containing lines of content, each given as a list of single statements
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
Notes
There needs to be at least one main keyword, otherwise the content is not added.
The content will be added at the end of the actual subkeyword.
If no sub keyword is present, a blank ("") will be added and the content is then directly connected
to the actual main keyword.

add_sub_keyword(self, key, main_index=None, sub_index=None)
Add a new sub keyword ($key) to the actual file.

Parameters
* key (string) – key name
* main_index (int, optional) – index of the corresponding main keyword
  where the sub keyword should be added. As default, the last main keyword is
  taken.
* sub_index (int, optional) – position, where the new sub keyword
  should be added between the existing ones. As default, it is placed at the end.

Notes
There needs to be at least one main keyword, otherwise the subkeyword is not added.

append_to_block(self, index=None, **block)
Append data to an existing Block in the actual file.
Keywords are the sub keywords of the actual file type:

#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...

which looks like the following:
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)

Parameters
* index (int or None, optional) – Positional index, where to insert the
  given Block. As default, it will be added at the end. Default: None.
* **block (keyword dict) – here the dict-keywords are the ogs-subkeywords
  and the value is the content that should be added with this ogs-subkeyword If a
  block should contain content directly connected to a main keyword, use this main
  keyword as input-keyword and the content as value: SUBKEY=content

check(self, verbose=True)
Check if the given file is valid.

Parameters verbose (bool, optional) – Print information for the executed
checks. Default: True

Returns result – Validity of the given file.
Return type bool

del_block(self, index=None, del_all=False)
Delete a block by its index.

Parameters
del_content (self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)
Delete content by its position.

Parameters

- **main_index** (int, optional) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- **sub_index** (int, optional) – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.
- **line_index** (int, optional) – position of the content line, that should be deleted. Default: -1
- **del_all** (bool, optional) – State, if all content shall be deleted. Default: False

del_copy_link (self)
Remove a former given link to an external file.

del_main_keyword (self, main_index=None, del_all=False)
Delete a main keyword (#key) by its position.

Parameters

- **main_index** (int, optional) – position, which main keyword should be deleted. Default: -1
- **del_all** (bool, optional) – State, if all main keywords shall be deleted. Default: False

del_sub_keyword (self, main_index=-1, sub_index=-1, del_all=False)
Delete a sub keyword ($key) by its position.

Parameters

- **main_index** (int, optional) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- **pos** (int, optional) – position, which sub keyword should be deleted. Default: -1
- **del_all** (bool, optional) – State, if all sub keywords shall be deleted. Default: False

get_block (self, index=None, as_dict=True)
Get a Block from the actual file.

Parameters

- **index** (int or None, optional) – Positional index of the block of interest. As default, the last one is returned. Default: None
- **as_dict** (bool, optional) – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for add_block. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

get_block_no (self)
Get the number of blocks in the file.

get_file_type (self)
Get the OGS file class name.
get_multi_keys(self, index=None)
    State if a block has a unique set of sub keywords.

is_block_unique(self, index=None)
    State if a block has a unique set of sub keywords.

read_file(self, path, encoding=None, verbose=False)
    Read an existing OGS input file.

    Parameters
    • path (str) – path to the existing file that should be read
    • encoding (str or None, optional) – encoding of the given file. If
      None is given, the system standard is used. Default: None
    • verbose (bool, optional) – Print information of the reading process. De-
      fault: False

reset(self)
    Delete every content.

save(self, path, **kwargs)
    Save the actual OGS input file in the given path.

    Parameters
    • path (str) – path to where to file should be saved
    • update (bool, optional) – state if the content should be updated before
      saving. Default: True

update_block(self, index=None, main_key=None, **block)
    Update a Block from the actual file.

    Parameters
    • index (int or None, optional) – Positional index of the block of inter-
      est. As default, the last one is used. Default: None
    • main_key (string, optional) – Main keyword of the block that should
      be updated (see: MKEYS) This shouldn’t be done. Default: None
    • **block (keyword dict) – here the dict-keywords are the ogs-subkeywords
      and the value is the content that should be added with this ogs-subkeyword If a
      block should contain content directly connected to a main keyword, use this main
      keyword as input-keyword and the content as value: SUBKEY=content

write_file(self)
    Write the actual OGS input file to the given folder.

    Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['INITIAL_CONDITION']

SKEYS = [['PCS_TYPE', 'PRIMARY_VARIABLE', 'COMP_NAME', 'STORE_VALUES', 'DIS_TYPE', 'GEO_TYPE']

STD = {'DIS_TYPE': ['CONSTANT', 0.0], 'GEO_TYPE': 'DOMAIN', 'PCS_TYPE': 'GROUNDWATER'}

block_no
    Number of blocks in the file.

file_name
    base name of the file with extension.

    Type str

file_path
    save path of the file.

    Type str
force_writing
    state if the file is written even if empty.
    
    Type bool

is_empty
    State if the OGS file is empty.

name
    name of the file without extension.
    
    Type str

class RFR(variables=None, data=None, units=None, headers=None, name=None, file_ext='.rfr',
    task_root=None, task_id='model')
Bases: ogs5py.fileclasses.base.File

Class for the ogs RESTART file, if the DIS_TYPE in IC is set to RESTART.

Parameters

* variables (list of str, optional) – List of variable names. Default: None
* data (numpy.ndarray, optional) – RFR data. 2D array, where the first dimension is the number of variables. Default: None
* units (list of str, optional) – List of units for the occurring variables. Can be None. OGS5 ignores them anyway. Default: None
* headers (str or None, optional) – First four lines of the RFR file. If None, a standard header is written. Default: None
* name (str, optional) – File name for the RFR file. If None, the task_id is used. Default: None
* file_ext (str, optional) – extension of the file (with leading dot “.rfr”) Default: “.rfr”
* task_root (str, optional) – Path to the destiny model folder. Default: cwd+”ogs5model”
* task_id (str, optional) – Name for the ogs task. Default: “model”

Notes

First line (ignored):
    • #0#0#0#1#100000#0... (don’t ask why)

Second line (ignored):
    • 1 1 4 (don’t ask why)

Third line (information about Variables):
    • (No. of Var.) (No of data of 1. Var) (No of data of 2. Var) ...
    • 1 1 (example: 1 Variable with 1 component)
    • 2 1 1 (example: 2 Variables with 1 component each)
    • only 1 scalar per Variable allowed (bug in OGS5). See: https://github.com/ufz/ogs5/issues/151

Fourth line (Variable names and units):
    • (Name1), (Unit1), (Name2), (Unit2), ...
    • units are ignored

Data (multiple lines):
    • (index) (Var1data1) .. (Var1dataN1) (Var2data1) .. (Var2dataN2) ..
Attributes

- **data** Data in the RFR file.
- **file_name** str: base name of the file with extension.
- **file_path** str: save path of the file.
- **force_writing** bool: state if the file is written even if empty.
- **is_empty** State if the OGS file is empty.
- **name** str: name of the file without extension.
- **units** List of variable-units in the RFR file.
- **var_count** Count of variables in the RFR file (line 3).
- **var_info** Infos about variables and units in the RFR file (line 4).
- **variables** List of variables in the RFR file.

Methods

- **add_copy_link** (self, path[, symlink]) Add a link to copy a file instead of writing.

  Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

  **Parameters**

  - **path** (str) – path to the existing file that should be copied
  - **symlink** (bool, optional) – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

- **check** (self[, verbose]) Check if the external geometry definition is valid.

  In the sense, that the contained data is consistent.

  **Parameters**

  - **verbose** (bool, optional) – Print information for the executed checks. Default: True

  **Returns** result – Validity of the given gli.

  **Return type** bool

- **del_copy_link** (self) Remove a former given link to an external file.

- **get_file_type** (self) Get the OGS file class name.

- **read_file** (self, path[, encoding, verbose]) Write the actual RFR input file to the given folder.

- **reset** (self) Delete every content.

- **save** (self, path, **kwargs) Save the actual RFR external file in the given path.

- **write_file** (self) Write the actual OGS input file to the given folder.

```python
add_copy_link (self, path[, symlink=False])
    Add a link to copy a file instead of writing.

    Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

    **Parameters**

    - **path** (str) – path to the existing file that should be copied
    - **symlink** (bool, optional) – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

check (self, verbose=True)
    Check if the external geometry definition is valid.

    In the sense, that the contained data is consistent.

    **Parameters**

    - **verbose** (bool, optional) – Print information for the executed checks. Default: True

    **Returns** result – Validity of the given gli.

    **Return type** bool

del_copy_link (self)
    Remove a former given link to an external file.
```
get_file_type(self)
Get the OGS file class name.

read_file(self, path, encoding=None, verbose=False)
Write the actual RFR input file to the given folder.

reset(self)
Delete every content.

save(self, path, **kwargs)
Save the actual RFR external file in the given path.

Parameters path (str) – path to where to file should be saved

write_file(self)
Write the actual OGS input file to the given folder.
Its path is given by “task_root+task_id+file_ext”.

data
Data in the RFR file.

file_name
base name of the file with extension.

    Type str

file_path
save path of the file.

    Type str

force_writing
state if the file is written even if empty.

    Type bool

is_empty
State if the OGS file is empty.

name
name of the file without extension.

    Type str

units
List of variable-units in the RFR file.

var_count
Count of variables in the RFR file (line 3).

var_info
Infos about variables and units in the RFR file (line 4).

variables
List of variables in the RFR file.
ogs5py.fileclasses.krc

Class for the ogs KINETRIC REACTION file.

File Class

```
KRC(**OGS_Config) Class for the ogs KINETRIC REACTION file.
```

class KRC(**OGS_Config)
    Bases: ogs5py.fileclasses.base.BlockFile
    Class for the ogs KINETRIC REACTION file.

    Parameters
    • `task_root (str, optional)` – Path to the destiny model folder. Default: cwd+”ogs5model”
    • `task_id (str, optional)` – Name for the ogs task. Default: “model”

Notes

Main-Keywords (#):
  • MICROBE_PROPERTIES
  • REACTION
  • BLOB_PROPERTIES
  • KINREACTIONDATA

Sub-Keywords ($) per Main-Keyword:
  • MICROBE_PROPERTIES
    – MICROBENAME
    – _drmc__PARAMETERS
    – MONOD_REACTION_NAME
  • REACTION
    – NAME
    – TYPE
    – BACTERIANAME
    – EQUATION
    – RATECONSTANT
    – GROWTH
    – MONODTERMS
    – THRESHHOLDTERMS
    – INHIBITIONTERMS
    – PRODUCTIONTERMS
    – PRODUCTIONSTOCH
    – BACTERIAL_YIELD
- ISOTOPES
- BACTERIA_SPECIFIC_CAPACITY
- TEMPERATURE_DEPENDENCE
- _dmc_
- STANDARD_GIBBS_ENERGY
- EXCHANGE_PARAMETERS
- SORPTION_TYPE
- NAPL_PROPERTIES
- REACTION_ORDER
- MINERALNAME
- CHEMAPPNAME
- EQUILIBRIUM_CONSTANT
- RATE_EXPONENTS
- REACTIVE_SURFACE_AREA
- PRECIPITATION_BY_BASETERM_ONLY
- PRECIPITATION_FACTOR
- PRECIPITATION_EXPONENT
- BASETERM
- MECHANISMTERM
- SWITCH_OFF_GEOMETRY

• BLOB_PROPERTIES
  - NAME
  - D50
  - DM
  - DS
  - UI
  - NAPL_CONTENT_INI
  - NAPL_CONTENT_RES
  - GRAIN_SPHERE_RATIO
  - TORTUOSITY
  - LENGTH
  - CALC_SHERWOOD
  - CALC_SHERWOOD_MODIFIED
  - SHERWOOD_MODEL
  - GEOMETRY
  - GAS_DISSOLUTION
  - INTERFACIAL_AREA

• KINREACTIONDATA
  - SOLVER_TYPE
- RELATIVE_ERROR
- MIN_TIMESTEP
- INITIAL_TIMESTEP
- BACTERIACAPACITY
- MIN_BACTERIACONC
- MIN_CONCENTRATION_REPLACE
- SURFACES
- ALLOW_REACTIONS
- NO_REACTIONS
- COPY_CONCENTRATIONS
- LAGNEAU_BENCHMARK
- SCALE_DCDT
- SORT_NODES
- OMEGA_THRESHOLD
- REACTION_DEACTIVATION
- DEBUG_OUTPUT
- ACTIVITY_MODEL

**Standard block:** None

**Keyword documentation:** https://ogs5-keywords.netlify.com/ogs/wiki/public/doc-auto/by_ext/krc

**Reading routines:** https://github.com/ufz/ogs5/blob/master/FEM/rfc_kinreact.cpp

- REACTION : https://github.com/ufz/ogs5/blob/master/FEM/rfc_kinreact.cpp#L1549
- BLOB_PROPERTIES : https://github.com/ufz/ogs5/blob/master/FEM/rfc_kinreact.cpp#L2622
- KINREACTIONDATA : https://github.com/ufz/ogs5/blob/master/FEM/rfc_kinreact.cpp#L3185

**See also:**

- add_block

**Attributes**

- `block_no` Number of blocks in the file.
- `file_name` str: base name of the file with extension.
- `file_path` str: save path of the file.
- `force_writing` bool: state if the file is written even if empty.
- `is_empty` State if the OGS file is empty.
- `name` str: name of the file without extension.

**Methods**

```python
add_block(self[, index, main_key]) Add a new Block to the actual file.
add_content(self, content[, main_index, ...]) Add single-line content to the actual file.
```

Continued on next page

3.6. ogs5py.fileclasses 95
add_copy_link(self, path[, symlink])  Add a link to copy a file instead of writing.

add_main_keyword(self, key[, main_index])  Add a new main keyword (#key) to the actual file.

add_multi_content(self, content[, ...])  Add multiple content to the actual file.

add_sub_keyword(self, key[, main_index, ...])  Add a new sub keyword ($key) to the actual file.

append_to_block(self[, index])  Append data to an existing Block in the actual file.

check(self[, verbose])  Check if the given file is valid.

del_block(self, index, del_all)  Delete a block by its index.

del_content(self[, main_index, sub_index, ...])  Delete content by its position.

del_copy_link(self)  Remove a former given link to an external file.

del_main_keyword(self[, main_index, del_all])  Delete a main keyword (#key) by its position.

del_sub_keyword(self[, main_index, ...])  Delete a sub keyword ($key) by its position.

get_block(self[, index, as_dict])  Get a Block from the actual file.

get_block_no(self)  Get the number of blocks in the file.

get_file_type(self)  Get the OGS file class name.

get_multi_keys(self[, index])  State if a block has a unique set of sub keywords.

is_block_unique(self[, index])  State if a block has a unique set of sub keywords.

read_file(self, path[, encoding, verbose])  Read an existing OGS input file.

reset(self)  Delete every content.

save(self, path, **kwargs)  Save the actual OGS input file in the given path.

update_block(self[, index, main_key])  Update a Block from the actual file.

write_file(self)  Write the actual OGS input file to the given folder.

add_block (self, index=None, main_key=None, **block)
Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...

which looks like the following:

FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)

Parameters
- **index** (int or None, optional) – Positional index, where to insert the
given Block. As default, it will be added at the end. Default: None.
- **main_key** (string, optional) – Main keyword of the block that should
be added (see: MKEYS) Default: the first main keyword of the file-type
- **block** (keyword dict) – here the dict-keywords are the ogs-subkeywords
and the value is the content that should be added with this ogs-subkeyword
If a block should contain content directly connected to a main keyword, use this main
keyword as input-keyword and the content as value: SUBKEY=content

add_content (self, content, main_index=None, sub_index=None, line_index=None)
Add single-line content to the actual file.

Parameters
- **content** (list) – list containing one line of content given as a list of single
  statements
• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

• **line_index** *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**
There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

---

**add_copy_link**(self, path, symlink=False)
Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

**Parameters**

• **path** *(str)* – path to the existing file that should be copied

• **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword**(self, key, main_index=None)
Add a new main keyword (#key) to the actual file.

**Parameters**

• **key** *(string)* – key name

• **main_index** *(int, optional)* – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

**add_multi_content**(self, content, main_index=None, sub_index=None)
Add multiple content to the actual file.

**Parameters**

• **content** *(list)* – list containing lines of content, each given as a list of single statements

• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

**Notes**
There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

---

**add_sub_keyword**(self, key, main_index=None, sub_index=None)
Add a new sub keyword ($key) to the actual file.
Parameters

- **key** *(string)* – key name
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

Notes
There needs to be at least one main keyword, otherwise the subkeyword is not added.

`append_to_block(self, index=None, **block)`
Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

Parameters

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None
- **block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: `SUBKEY=content`

`check(self, verbose=True)`
Check if the given file is valid.

Parameters

- **verbose** *(bool, optional)* – Print information for the executed checks. Default: True

Returns

- **result** – Validity of the given file.

Return type

- **bool**

`del_block(self, index=None, del_all=False)`
Delete a block by its index.

Parameters

- **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None
- **del_all** *(bool, optional)* – State, if all blocks shall be deleted. Default: False

`del_content(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)`
Delete content by its position.

Parameters

- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
• **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.

• **line_index** *(int, optional)* – position of the content line, that should be deleted. Default: -1

• **del_all** *(bool, optional)* – State, if all content shall be deleted. Default: False

**del_copy_link**(self)
Remove a former given link to an external file.

**del_main_keyword**(self, main_index=None, del_all=False)
Delete a main keyword (#key) by its position.

Parameters

• **main_index** *(int, optional)* – position, which main keyword should be deleted. Default: -1

• **del_all** *(bool, optional)* – State, if all main keywords shall be deleted. Default: False

**del_sub_keyword**(self, main_index=-1, sub_index=-1, del_all=False)
Delete a sub keyword ($key) by its position.

Parameters

• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• **pos** *(int, optional)* – position, which sub keyword should be deleted. Default: -1

• **del_all** *(bool, optional)* – State, if all sub keywords shall be deleted. Default: False

**get_block**(self, index=None, as_dict=True)
Get a Block from the actual file.

Parameters

• **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

• **as_dict** *(bool, optional)* – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for add_block. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no**(self)
Get the number of blocks in the file.

**get_file_type**(self)
Get the OGS file class name.

**get_multi_keys**(self, index=None)
State if a block has a unique set of sub keywords.

**is_block_unique**(self, index=None)
State if a block has a unique set of sub keywords.

**read_file**(self, path, encoding=None, verbose=False)
Read an existing OGS input file.

Parameters

• **path** *(str)* – path to the existing file that should be read
• **encoding** (*str or None, optional*) – encoding of the given file. If
  *None* is given, the system standard is used. Default: *None*

• **verbose** (*bool, optional*) – Print information of the reading process. De-
  fault: *False*

```
reset (self)
Delete every content.
```

```
save (self, path, **kwargs)
Save the actual OGS input file in the given path.
Parameters
• **path** (*str*) – path to where to file should be saved
• **update** (*bool, optional*) – state if the content should be updated before
  saving. Default: *True*
```

```
update_block (self, index=None, main_key=None, **block)
Update a Block from the actual file.
Parameters
• **index** (*int or None, optional*) – Positional index of the block of inter-
  est. As default, the last one is used. Default: *None*
• **main_key** (*string, optional*) – Main keyword of the block that should
  be updated (see: MKEYS) This shouldn’t be done. Default: *None*
• **block** (*keyword dict*) – here the dict-keywords are the ogs-subkeywords
  and the value is the content that should be added with this ogs-subkeyword If a
  block should contain content directly connected to a main keyword, use this main
  keyword as input-keyword and the content as value: SUBKEY=content
```

```
write_file (self)
Write the actual OGS input file to the given folder.
Its path is given by “task_root+task_id+file_ext”.
```

```
MKEYS = ['MICROBE_PROPERTIES', 'REACTION', 'BLOB_PROPERTIES', 'KINREACTIONDATA']
SKEYS = [['MICROBENAME', '_drmc__PARAMETERS', 'MONOD_REACTION_NAME'], ['NAME', 'TYPE', 'BACTERIANAME', 'EQUATION', ...
  'OMEGA_THRESHOLD', 'REACTION_DEACTIVATION', 'DEBUG_OUTPUT', 'ACTIVITY_MODEL', 'REALATIVE_ERROR', 'MAX_TIMESTEP']
STD = {}
```

```
block_no
Number of blocks in the file.
```

```
file_name
base name of the file with extension.
  Type *str*
```

```
file_path
save path of the file.
  Type *str*
```

```
force_writing
state if the file is written even if empty.
  Type *bool*
```

```
is_empty
State if the OGS file is empty.
```

```
name
name of the file without extension.
  Type *str*
```
ogs5py.fileclasses.mcp

Class for the ogs COMPONENT_PROPERTIES file.

File Class

```python
MCP(**OGS_Config)  # Class for the ogs COMPONENT_PROPERTIES file.
```

class MCP(**OGS_Config)
Bases: ogs5py.fileclasses.base.BlockFile

Class for the ogs COMPONENT_PROPERTIES file.

Parameters

- **task_root** *(str, optional)* – Path to the destiny model folder. Default: cwd+”ogs5model”
- **task_id** *(str, optional)* – Name for the ogs task. Default: “model”

Notes

Main-Keywords (#):

- COMPONENT_PROPERTIES

Sub-Keywords ($) per Main-Keyword:

- COMPONENT_PROPERTIES
  - ACENTRIC_FACTOR
  - A_ZERO
  - BUBBLE_VELOCITY
  - CRITICAL_PRESSURE
  - CRITICAL_TEMPERATURE
  - DECAY
  - DIFFUSION
  - FLUID_ID
  - FLUID_PHASE
  - FORMULA
  - ISOTHERM
  - MAXIMUM_AQUEOUS_SOLUBILITY
  - MINERAL_DENSITY
  - MOBILE
  - MOLAR_DENSITY
  - MOLAR_VOLUME
  - MOLAR_WEIGHT
  - MOL_MASS
  - NAME
– OutputMassOfComponentInModel
– TRANSPORT_PHASE
– VALENCE
– VOLUME_DIFFUSION

Standard block: None


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rfmat_cp.cpp#L269

See also:

add_block

Attributes

block_no Number of blocks in the file.

file_name str: base name of the file with extension.

guideline str: save path of the file.

force_writing bool: state if the file is written even if empty.

is_empty State if the OGS file is empty.

name str: name of the file without extension.

Methods

add_block(self[, index, main_key]) Add a new Block to the actual file.

add_content(self, content[, main_index, ...]) Add single-line content to the actual file.

add_copy_link(self, path[, symlink]) Add a link to copy a file instead of writing.

add_main_keyword(self, key[, main_index]) Add a new main keyword (#key) to the actual file.

add_multi_content(self, content[, ...]) Add multiple content to the actual file.

add_sub_keyword(self, key[, main_index, ...]) Add a new sub keyword ($key) to the actual file.

append_to_block(self[, index]) Append data to an existing Block in the actual file.

check(self[, verbose]) Check if the given file is valid.

del_block(self[, index, del_all]) Delete a block by its index.

del_content(self[, main_index, sub_index, ...]) Delete content by its position.

del_copy_link(self) Remove a former given link to an external file.

del_main_keyword(self[, main_index, del_all]) Delete a main keyword (#key) by its position.

del_sub_keyword(self[, main_index, ...]) Delete a sub keyword ($key) by its position.

get_block(self[, index, as_dict]) Get a Block from the actual file.

get_block_no(self) Get the number of blocks in the file.

get_file_type(self) Get the OGS file class name.

get_multi_keys(self[, index]) State if a block has a unique set of sub keywords.

is_block_unique(self[, index]) State if a block has a unique set of sub keywords.

read_file(self, path[, encoding, verbose]) Read an existing OGS input file.

reset(self) Delete every content.

save(self, path, **kwargs) Save the actual OGS input file in the given path.

update_block(self[, index, main_key]) Update a Block from the actual file.

write_file(self) Write the actual OGS input file to the given folder.
**add_block** *(self, index=None, main_key=None, **block)*

Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

#MAIN_KEY

$SUBKEY1 content1...

$SUBKEY2 content2 ...

which looks like the following:

FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)

**Parameters**

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

- **main_key** *(string, optional)* – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type

- ****block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**add_content** *(self, content, main_index=None, sub_index=None, line_index=None)*

Add single-line content to the actual file.

**Parameters**

- **content** *(list)* – list containing one line of content given as a list of single statements

- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

- **line_index** *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

**add_copy_link** *(self, path, symlink=False)*

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linking to the target folder.

**Parameters**

- **path** *(str)* – path to the existing file that should be copied

- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword** *(self, key, main_index=None)*

Add a new main keyword (#key) to the actual file.
add_multi_content (self, content, main_index=None, sub_index=None)
Add multiple content to the actual file.

Parameters

- **content (list)** – list containing lines of content, each given as a list of single statements
- **main_index (int, optional)** – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index (int, optional)** – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

Notes

There needs to be at least one main keyword, otherwise the content is not added.
The content will be added at the end of the actual subkeyword.
If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

add_sub_keyword (self, key, main_index=None, sub_index=None)
Add a new sub keyword ($key) to the actual file.

Parameters

- **key (string)** – key name
- **main_index (int, optional)** – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index (int, optional)** – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

Notes

There needs to be at least one main keyword, otherwise the subkeyword is not added.

append_to_block (self, index=None, **block)
Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

#MAIN_KEY

$SUBKEY1 content1 ...
$SUBKEY2 content2 ...

which looks like the following:

FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)

Parameters

- **index (int or None, optional)** – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
**block** (`keyword dict`) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: `SUBKEY=content`

```python
check(self, verbose=True)
```
Check if the given file is valid.

**Parameters**

`verbose` (`bool`, optional) – Print information for the executed checks. Default: True

**Returns**

`result` – Validity of the given file.

**Return type**

`bool`

```python
del_block(self, index=None, del_all=False)
```
Delete a block by its index.

**Parameters**

- `index` (`int` or `None`, optional) – Positional index of the block of interest. As default, the last one is returned. Default: None
- `del_all` (`bool`, optional) – State, if all blocks shall be deleted. Default: False

```python
del_content(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)
```
Delete content by its position.

**Parameters**

- `main_index` (`int`, optional) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- `sub_index` (`int`, optional) – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.
- `line_index` (`int`, optional) – position of the content line, that should be deleted. Default: -1
- `del_all` (`bool`, optional) – State, if all content shall be deleted. Default: False

```python
del_copy_link(self)
```
Remove a former given link to an external file.

```python
del_main_keyword(self, main_index=None, del_all=False)
```
Delete a main keyword (#key) by its position.

**Parameters**

- `main_index` (`int`, optional) – position, which main keyword should be deleted. Default: -1
- `del_all` (`bool`, optional) – State, if all main keywords shall be deleted. Default: False

```python
del_sub_keyword(self, main_index=-1, sub_index=-1, del_all=False)
```
Delete a sub keyword ($key) by its position.

**Parameters**

- `main_index` (`int`, optional) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- `pos` (`int`, optional) – position, which sub keyword should be deleted. Default: -1
• **del_all** *(bool, optional)* – State, if all sub keywords shall be deleted. Default: False

**get_block** *(self, index=None, as_dict=True)*
Get a Block from the actual file.

**Parameters**

• **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

• **as_dict** *(bool, optional)* – Here you can state if you want the output as a dictionary, which can be used as key-word-arguments for **add_block**. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no** *(self)*
Get the number of blocks in the file.

**get_file_type** *(self)*
Get the OGS file class name.

**get_multi_keys** *(self, index=None)*
State if a block has a unique set of sub keywords.

**is_block_unique** *(self, index=None)*
State if a block has a unique set of sub keywords.

**read_file** *(self, path, encoding=None, verbose=False)*
Read an existing OGS input file.

**Parameters**

• **path** *(str)* – path to the existing file that should be read

• **encoding** *(str or None, optional)* – encoding of the given file. If None is given, the system standard is used. Default: None

• **verbose** *(bool, optional)* – Print information of the reading process. Default: False

**reset** *(self)*
Delete every content.

**save** *(self, path, **kwargs)*
Save the actual OGS input file in the given path.

**Parameters**

• **path** *(str)* – path to where to file should be saved

• **update** *(bool, optional)* – state if the content should be updated before saving. Default: True

**update_block** *(self, index=None, main_key=None, **block)*
Update a Block from the actual file.

**Parameters**

• **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is used. Default: None

• **main_key** *(string, optional)* – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None

• **block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content
write_file(self)
    Write the actual OGS input file to the given folder.
    Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['COMPONENT_PROPERTIES']
SKEYS = [['NAME', 'FORMULA', 'MOBILE', 'TRANSPORT_PHASE', 'FLUID_PHASE', 'MOL_MASS', ...
STD = {}

block_no
    Number of blocks in the file.

file_name
    base name of the file with extension.
    Type str

file_path
    save path of the file.
    Type str

force_writing
    state if the file is written even if empty.
    Type bool

is_empty
    State if the OGS file is empty.

name
    name of the file without extension.
    Type str
ogspyDocumentation, Release 1.1.1

ogspy.fileclasses.mfp

Class for the ogs FLUID PROPERTY file.

### File Class

```python
MFP(**OGS_Config) Class for the ogs FLUID PROPERTY file.
```

```python
class MFP (**OGS_Config)
    Bases: ogspy.fileclasses.base.BlockFile
    Class for the ogs FLUID PROPERTY file.

    Parameters
    • task_root (str, optional) – Path to the destiny model folder. Default: cwd+"ogs5model"
    • task_id (str, optional) – Name for the ogs task. Default: “model”
```

### Notes

**Main-Keywords (#):**

- FLUID_PROPERTIES

**Sub-Keywords ($) per Main-Keyword:**

- FLUID_PROPERTIES
  - COMPONENTS
  - COMPRESSIBILITY
  - DAT_TYPE
  - DECAY
  - DENSITY
  - DIFFUSION
  - DRHO_DT_UNSATURATED
  - EOS_TYPE
  - FLUID_NAME
  - FLUID_TYPE
  - GRAVITY
  - HEAT_CONDUCTIVITY
  - ISOTHERM
  - JTC
  - NON_GRAVITY
  - PHASE_DIFFUSION
  - SPECIFIC_HEAT_CAPACITY
  - SPECIFIC_HEAT_SOURCE
  - TEMPERATURE
- VISCOSITY

Standard block:

**FLUID_TYPE** “LIQUID”

**DENSITY** [1, 1.0e+03]

**VISCOSITY** [1, 1.0e-03]


See also:

* add_block

Attributes

- **block_no** Number of blocks in the file.
- **file_name** str: base name of the file with extension.
- **file_path** str: save path of the file.
- **force_writing** bool: state if the file is written even if empty.
- **is_empty** State if the OGS file is empty.
- **name** str: name of the file without extension.

Methods

- **add_block**(self[, index, main_key]) Add a new Block to the actual file.
- **add_content**(self, content[, main_index,...]) Add single-line content to the actual file.
- **add_copy_link**(self, path[, symlink]) Add a link to copy a file instead of writing.
- **add_main_keyword**(self, key[, main_index]) Add a new main keyword (#key) to the actual file.
- **add_multi_content**(self, content[, ...]) Add multiple content to the actual file.
- **add_sub_keyword**(self[, key[, main_index,...]]) Add a new sub keyword ($key) to the actual file.
- **append_to_block**(self[, index]) Append data to an existing Block in the actual file.
- **check**(self[, verbose]) Check if the given file is valid.
- **del_block**(self[, index, del_all]) Delete a block by its index.
- **del_content**(self[, main_index, sub_index,...]) Delete content by its position.
- **del_copy_link**(self) Remove a former given link to an external file.
- **del_main_keyword**(self[, main_index, del_all]) Delete a main keyword (#key) by its position.
- **del_sub_keyword**(self[, main_index,...]) Delete a sub keyword ($key) by its position.
- **get_block**(self[, index]) Get a Block from the actual file.
- **get_block_no**(self) Get the number of blocks in the file.
- **get_file_type**(self) Get the OGS file class name.
- **get_multi_keys**(self[, index]) State if a block has a unique set of sub keywords.
- **is_block_unique**(self[, index]) State if a block has a unique set of sub keywords.
- **read_file**(self, path[, encoding, verbose]) Read an existing OGS input file.
- **reset**(self) Delete every content.
- **save**(self, path[, \*\*kwargs]) Save the actual OGS input file in the given path.
- **update_block**(self[, index, main_key]) Update a Block from the actual file.
- **write_file**(self) Write the actual OGS input file to the given folder.
add_block(self, index=None, main_key=None, **block)
Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

#MAIN_KEY
$SUBKEY1 content1...
$SUBKEY2 content2...

which looks like the following:

FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)

Parameters

• index (int or None, optional) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

• main_key (string, optional) – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type

• **block (keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

add_content(self, content, main_index=None, sub_index=None, line_index=None)
Add single-line content to the actual file.

Parameters

• content (list) – list containing one line of content given as a list of single statements

• main_index (int, optional) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• sub_index (int, optional) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

• line_index (int, optional) – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

Notes
There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

add_copy_link(self, path, symlink=False)
Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/link to the target folder.

Parameters

• path (str) – path to the existing file that should be copied

• symlink (bool, optional) – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

add_main_keyword(self, key, main_index=None)
Add a new main keyword (#key) to the actual file.
Parameters

- key (string) – key name
- main_index (int, optional) – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

add_multi_content (self, content, main_index=None, sub_index=None)
Add multiple content to the actual file.

Parameters

- content (list) – list containing lines of content, each given as a list of single statements
- main_index (int, optional) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- sub_index (int, optional) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

Notes
There needs to be at least one main keyword, otherwise the content is not added.
The content will be added at the end of the actual subkeyword.
If no sub keyword is present, a blank one (“”) will be added and the content is then directly connected to the actual main keyword.

add_sub_keyword (self, key, main_index=None, sub_index=None)
Add a new sub keyword ($key) to the actual file.

Parameters

- key (string) – key name
- main_index (int, optional) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- sub_index (int, optional) – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

Notes
There needs to be at least one main keyword, otherwise the subkeyword is not added.

append_to_block (self, index=None, **block)
Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...

which looks like the following:
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)

Parameters

- index (int or None, optional) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
**block** (*keyword dict*) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: **SUBKEY=content**

```python
check(self, verbose=True)
```
Check if the given file is valid.

**Parameters**
* verbose (bool, optional) – Print information for the executed checks. Default: True

**Returns**
* result – Validity of the given file.

**Return type** bool

```python
del_block(self, index=None, del_all=False)
```
Delete a block by its index.

**Parameters**
* index (int or None, optional) – Positional index of the block of interest. As default, the last one is returned. Default: None
* del_all (bool, optional) – State, if all blocks shall be deleted. Default: False

```python
del_content(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)
```
Delete content by its position.

**Parameters**
* main_index (int, optional) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
* sub_index (int, optional) – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.
* line_index (int, optional) – position of the content line, that should be deleted. Default: -1
* del_all (bool, optional) – State, if all content shall be deleted. Default: False

```python
del_copy_link(self)
```
Remove a former given link to an external file.

```python
del_main_keyword(self, main_index=None, del_all=False)
```
Delete a main keyword (#key) by its position.

**Parameters**
* main_index (int, optional) – position, which main keyword should be deleted. Default: -1
* del_all (bool, optional) – State, if all main keywords shall be deleted. Default: False

```python
del_sub_keyword(self, main_index=-1, sub_index=-1, del_all=False)
```
Delete a sub keyword ($key) by its position.

**Parameters**
* main_index (int, optional) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
* pos (int, optional) – position, which sub keyword should be deleted. Default: -1
• **del_all** *(bool, optional)* – State, if all sub keywords shall be deleted. Default: False

**get_block** *(self, index=None, as_dict=True)*
Get a Block from the actual file.

**Parameters**

• **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

• **as_dict** *(bool, optional)* – Here you can state if you want the output as a dictionary, which can be used as key-word-arguments for **add_block**. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no** *(self)*
Get the number of blocks in the file.

**get_file_type** *(self)*
Get the OGS file class name.

**get_multi_keys** *(self, index=None)*
State if a block has a unique set of sub keywords.

**is_block_unique** *(self, index=None)*
State if a block has a unique set of sub keywords.

**read_file** *(self, path, encoding=None, verbose=False)*
Read an existing OGS input file.

**Parameters**

• **path** *(str)* – path to the existing file that should be read

• **encoding** *(str or None, optional)* – encoding of the given file. If None is given, the system standard is used. Default: None

• **verbose** *(bool, optional)* – Print information of the reading process. Default: False

**reset** *(self)*
Delete every content.

**save** *(self, path, **kwargs)*
Save the actual OGS input file in the given path.

**Parameters**

• **path** *(str)* – path to where to file should be saved

• **update** *(bool, optional)* – state if the content should be updated before saving. Default: True

**update_block** *(self, index=None, main_key=None, **block)*
Update a Block from the actual file.

**Parameters**

• **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is used. Default: None

• **main_key** *(string, optional)* – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None

• **block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: **SUBKEY=content**
write_file(self)
    Write the actual OGS input file to the given folder.
    Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['FLUID_PROPERTIES']
SKEYS = [['FLUID_TYPE', 'COMPONENTS', 'FLUID_NAME', 'EOS_TYPE', 'COMPRESSIBILITY', 'JTC', 'DAT_TYPE', 'NON_GRAVITY', ...
          'PHASE_DIFFUSION', 'DIFFUSION', 'DECAY', 'ISOTHERM', 'GRAVITY', 'SPECIFIC_HEAT_SOURCE', 'PCS_TYPE', 'THERMAL']
STD = {'DENSITY': [1, 1000.0], 'FLUID_TYPE': 'LIQUID', 'VISCOSITY': [1, 0.001]}

block_no
    Number of blocks in the file.

file_name
    base name of the file with extension.
    Type str

file_path
    save path of the file.
    Type str

force_writing
    state if the file is written even if empty.
    Type bool

is_empty
    State if the OGS file is empty.

name
    name of the file without extension.
    Type str
ogs5py.fileclasses.mmp

Class for the ogs MEDIUM_PROPERTIES file.

File Class

```python
MMP(**OGS_Config)  # Class for the ogs MEDIUM_PROPERTIES file.
```

class MMP(**OGS_Config):
    Bases: ogs5py.fileclasses.base.BlockFile

    Class for the ogs MEDIUM_PROPERTIES file.

    Parameters
    • `task_root (str, optional)` – Path to the destiny model folder. Default: cwd+"ogs5model"
    • `task_id (str, optional)` – Name for the ogs task. Default: “model”

Notes

Main-Keywords (#):
• MEDIUM_PROPERTIES

Sub-Keywords ($) per Main-Keyword:
• MEDIUM_PROPERTIES
  – CAPILLARY_PRESSURE
  – CHANNEL
  – COMPOUND_DEPENDENT_DT
  – CONDUCTIVITY_MODEL
  – CONVERSION_FACTOR
  – DATA
  – DIFFUSION
  – DIS_TYPE
  – ELEMENT_VOLUME_MULTIPLYER
  – EVAPORATION
  – FLOWLINEARITY
  – GEOMETRY_AREA
  – GEOMETRY_DIMENSION
  – GEOMETRY_INCLINATION
  – GEO_TYPE
  – HEAT_DISPERSION
  – HEAT_TRANSFER
  – INTERPHASE_FRICTION
  – MASS_DISPERSION

3.6. ogs5py.fileclasses
- MMP_TYPE
- MSH_TYPE
- NAME
- ORGANIC_CARBON
- PARTICLE_DIAMETER
- PCS_TYPE
- PERMEABILITY_FUNCTION_DEFORMATION
- PERMEABILITY_FUNCTION_EFFSTRESS
- PERMEABILITY_FUNCTION_POROSITY
- PERMEABILITY_FUNCTION_PRESSURE
- PERMEABILITY_FUNCTION_STRAIN
- PERMEABILITY_FUNCTION_STRESS
- PERMEABILITY_FUNCTION_VELOCITY
- PERMEABILITY_SATURATION
- PERMEABILITY_TENSOR
- PERMEABILITY_DISTRIBUTION
- POROSITY
- POROSITY_DISTRIBUTION
- RILL
- SPECIFIC_STORAGE
- STORAGE
- STORAGE_FUNCTION_EFFSTRESS
- SURFACE_FRICTION
- TORTUOSITY
- TRANSFER_COEFFICIENT
- UNCONFINED
- VOL_BIO
- VOL_MAT
- WIDTH

Standard block:

```plaintext
GEOMETRY_DIMENSION 2,
STORAGE [1, 1.0e-4],
PERMEABILITY_TENSOR ["ISOTROPIC", 1.0e-4],
POROSITY [1, 0.2]
```


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rf_mmp_new.cpp#L281

See also:

add_block
Attributes

- **block_no** Number of blocks in the file.
- **file_name** str: base name of the file with extension.
- **file_path** str: save path of the file.
- **force_writing** bool: state if the file is written even if empty.
- **is_empty** State if the OGS file is empty.
- **name** str: name of the file without extension.

Methods

```python
add_block(self[, index, main_key]) Add a new Block to the actual file.
add_content(self, content[, main_index, ...]) Add single-line content to the actual file.
add_copy_link(self[, path[, symlink]]) Add a link to copy a file instead of writing.
add_main_keyword(self[, key[, main_index]]) Add a new main keyword (#key) to the actual file.
add_multi_content(self[, content[, ...]]) Add multiple content to the actual file.
add_sub_keyword(self[, key[, main_index, ...]]) Add a new sub keyword ($key) to the actual file.
append_to_block(self[, index]) Append data to an existing Block in the actual file.
check(self[, verbose]) Check if the given file is valid.
del_block(self[, index, del_all]) Delete a block by its index.
del_content(self[, main_index, sub_index, ...]) Delete content by its position.
del_copy_link(self) Remove a former given link to an external file.
del_main_keyword(self[, main_index, del_all]) Delete a main keyword (#key) by its position.
del_sub_keyword(self[, main_index, ...]) Delete a sub keyword ($key) by its position.
get_block(self[, index, as_dict]) Get a Block from the actual file.
get_block_no(self) Get the number of blocks in the file.
get_file_type(self) Get the OGS file class name.
get_multi_keys(self[, index]) State if a block has a unique set of sub keywords.
is_block_unique(self[, index]) State if a block has a unique set of sub keywords.
read_file(self[, path[, encoding, verbose]]) Read an existing OGS input file.
reset(self) Delete every content.
save(self, path, **kwargs) Save the actual OGS input file in the given path.
update_block(self[, index, main_key]) Update a Block from the actual file.
write_file(self) Write the actual OGS input file to the given folder.
```

```python
add_block(self, index=None, main_key=None, **block)
```

Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

```text
#MAIN_KEY

$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```python
FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)
```

Parameters

- **index** (*int or None, optional*) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
• **main_key**(string, optional) – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type

• **block**(keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**add_content**(self, content, main_index=None, sub_index=None, line_index=None)
Add single-line content to the actual file.

Parameters

• **content**(list) – list containing one line of content given as a list of single statements

• **main_index**(int, optional) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index**(int, optional) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

• **line_index**(int, optional) – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**
There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

**add_copy_link**(self, path, symlink=False)
Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

Parameters

• **path**(str) – path to the existing file that should be copied

• **symlink**(bool, optional) – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword**(self, key, main_index=None)
Add a new main keyword (#key) to the actual file.

Parameters

• **key**(string) – key name

• **main_index**(int, optional) – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

**add_multi_content**(self, content, main_index=None, sub_index=None)
Add multiple content to the actual file.

Parameters

• **content**(list) – list containing lines of content, each given as a list of single statements

• **main_index**(int, optional) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
• **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.
The content will be added at the end of the actual subkeyword.
If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

### add_sub_keyword

```python
add_sub_keyword(self, key, main_index=None, sub_index=None)
```

Add a new sub keyword ($key) to the actual file.

**Parameters**

- **key**(string) – key name
- **main_index**(int, optional) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index**(int, optional) – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the subkeyword is not added.

### append_to_block

```python
append_to_block(self, index=None, **block)
```

Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- **index**(int or None, optional) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- ****block**(keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

### check

```python
check(self, verbose=True)
```

Check if the given file is valid.

**Parameters** **verbose**(bool, optional) – Print information for the executed checks. Default: True

**Returns** result – Validity of the given file.

**Return type** bool

### del_block

```python
del_block(self, index=None, del_all=False)
```

Delete a block by its index.
Parameters

- **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

- **del_all** *(bool, optional)* – State, if all blocks shall be deleted. Default: False

**del_content**(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)
Delete content by its position.

Parameters

- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.

- **line_index** *(int, optional)* – position of the content line, that should be deleted. Default: -1

- **del_all** *(bool, optional)* – State, if all content shall be deleted. Default: False

**del_copy_link***(self)*
Remove a former given link to an external file.

**del_main_keyword**(self, main_index=None, del_all=False)
Delete a main keyword (#key) by its position.

Parameters

- **main_index** *(int, optional)* – position, which main keyword should be deleted. Default: -1

- **del_all** *(bool, optional)* – State, if all main keywords shall be deleted. Default: False

**del_sub_keyword**(self, main_index=-1, sub_index=-1, del_all=False)
Delete a sub keyword ($key) by its position.

Parameters

- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

- **pos** *(int, optional)* – position, which sub keyword should be deleted. Default: -1

- **del_all** *(bool, optional)* – State, if all sub keywords shall be deleted. Default: False

**get_block**(self, index=None, as_dict=True)
Get a Block from the actual file.

Parameters

- **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

- **as_dict** *(bool, optional)* – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for add_block. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no**(self)
Get the number of blocks in the file.
get_file_type (self)
  Get the OGS file class name.

get_multi_keys (self, index=None)
  State if a block has a unique set of sub keywords.

is_block_unique (self, index=None)
  State if a block has a unique set of sub keywords.

read_file (self, path, encoding=None, verbose=False)
  Read an existing OGS input file.

  Parameters

  • path (str) – path to the existing file that should be read
  • encoding (str or None, optional) – encoding of the given file. If
    None is given, the system standard is used. Default: None
  • verbose (bool, optional) – Print information of the reading process. De-
    fault: False

reset (self)
  Delete every content.

save (self, path, **kwargs)
  Save the actual OGS input file in the given path.

  Parameters

  • path (str) – path to where to file should be saved
  • update (bool, optional) – state if the content should be updated before
    saving. Default: True

update_block (self, index=None, main_key=None, **block)
  Update a Block from the actual file.

  Parameters

  • index (int or None, optional) – Positional index of the block of inter-
    est. As default, the last one is used. Default: None
  • main_key (string, optional) – Main keyword of the block that should
    be updated (see: MKEYS) This shouldn’t be done. Default: None
  • **block (keyword dict) – here the dict-keywords are the ogs-subkeywords
    and the value is the content that should be added with this ogs-subkeyword If a
    block should contain content directly connected to a main keyword, use this main
    keyword as input-keyword and the content as value: SUBKEY=content

write_file (self)
  Write the actual OGS input file to the given folder.

  Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['MEDIUM_PROPERTIES']

SKEYS = [['PCS_TYPE', 'NAME', 'GEO_TYPE', 'GEOMETRY_DIMENSION', 'GEOMETRY_INCLINATION', 'GEOMETRY_AREA', 'POROSITY', 'HEAT_TRANSFER', 'PARTICLE_DIAMETER', 'INTERPHASE_FRICTION', 'ELEMENT_VOLUME_MULTIPLYER', 'MEDIUM_TYPE', 'DENSITY']

STD = {'GEOMETRY_DIMENSION': 2, 'PERMEABILITY_TENSOR': ['ISOTROPIC', 0.0001], 'POROS'}

block_no
  Number of blocks in the file.

file_name
  base name of the file with extension.

  Type str
file_path
    save path of the file.
    Type  str

force_writing
    state if the file is written even if empty.
    Type  bool

is_empty
    State if the OGS file is empty.

name
    name of the file without extension.
    Type  str
**ogs5py.fileclasses.mpd**

Class for the ogs MEDIUM_PROPERTIES_DISTRIBUTED file.

**File Class**

```python
class MPD([name, file_ext])
```

Class for the ogs MEDIUM_PROPERTIES_DISTRIBUTED file.

**Parameters**

- **name** *(str, optional)* – File name for the MPD file. If None, the task_id is used. Default: None

- **file_ext** *(str, optional)* – Extension of the file (with leading dot “.mpd”) Default: “.mpd”

- **task_root** *(str, optional)* – Path to the destiny model folder. Default: cwd+”ogs5model”

- **task_id** *(str, optional)* – Name for the ogs task. Default: “model”

**Notes**

Main-Keywords (#):

- MEDIUM_PROPERTIES_DISTRIBUTED

Sub-Keywords ($) per Main-Keyword:

- MEDIUM_PROPERTIES_DISTRIBUTED
  - MSH_TYPE
  - MMP_TYPE
  - DIS_TYPE
  - CONVERSION_FACTOR
  - DATA

**Standard block:** None


**See also:**

`add_block`

**Attributes**

- **block_no** Number of blocks in the file.

- **file_name** *(str)*: base name of the file with extension.

- **file_path** *(str)*: save path of the file.
**force_writing** bool: state if the file is written even if empty.

**is_empty** State if the OGS file is empty.

**name** str: name of the file without extension.

**top_com** Top comment is ‘None’ for the MPD file.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add_block</strong>(self, index, main_key)</td>
<td>Add a new Block to the actual file.</td>
</tr>
<tr>
<td><strong>add_content</strong>(self, content[, main_index,...])</td>
<td>Add single-line content to the actual file.</td>
</tr>
<tr>
<td><strong>add_copy_link</strong>(self, path[, symlink])</td>
<td>Add a link to copy a file instead of writing.</td>
</tr>
<tr>
<td><strong>add_main_keyword</strong>(self, key[, main_index])</td>
<td>Add a new main keyword (#key) to the actual file.</td>
</tr>
<tr>
<td><strong>add_multi_content</strong>(self, content[, ...])</td>
<td>Add multiple content to the actual file.</td>
</tr>
<tr>
<td><strong>add_sub_keyword</strong>(self, key[, main_index,...])</td>
<td>Add a new sub keyword ($key) to the actual file.</td>
</tr>
<tr>
<td><strong>append_to_block</strong>(self[, index])</td>
<td>Append data to an existing Block in the actual file.</td>
</tr>
<tr>
<td><strong>check</strong>(self[, verbose])</td>
<td>Check if the given file is valid.</td>
</tr>
<tr>
<td><strong>del_block</strong>(self, index, del_all)</td>
<td>Delete a block by its index.</td>
</tr>
<tr>
<td><strong>del_content</strong>(self[, main_index, sub_index,...])</td>
<td>Delete content by its position.</td>
</tr>
<tr>
<td><strong>del_copy_link</strong>(self)</td>
<td>Remove a former given link to an external file.</td>
</tr>
<tr>
<td><strong>del_main_keyword</strong>(self[, main_index, del_all])</td>
<td>Delete a main keyword (#key) by its position.</td>
</tr>
<tr>
<td><strong>del_sub_keyword</strong>(self[, main_index,...])</td>
<td>Delete a sub keyword ($key) by its position.</td>
</tr>
<tr>
<td><strong>get_block</strong>(self[, index, as_dict])</td>
<td>Get a Block from the actual file.</td>
</tr>
<tr>
<td><strong>get_block_no</strong>(self)</td>
<td>Get the number of blocks in the file.</td>
</tr>
<tr>
<td><strong>get_file_type</strong>(self)</td>
<td>Get the OGS file class name.</td>
</tr>
<tr>
<td><strong>get_multi_keys</strong>(self[, index])</td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td><strong>is_block_unique</strong>(self[, index])</td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td><strong>read_file</strong>(self, path[, encoding, verbose])</td>
<td>Read an existing OGS input file.</td>
</tr>
<tr>
<td><strong>reset</strong>(self)</td>
<td>Delete every content.</td>
</tr>
<tr>
<td><strong>save</strong>(self, path, **kwargs)</td>
<td>Save the actual OGS input file in the given path.</td>
</tr>
<tr>
<td><strong>update_block</strong>(self, index, main_key)</td>
<td>Update a Block from the actual file.</td>
</tr>
<tr>
<td><strong>write_file</strong>(self)</td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>

**add_block**(self, index=None, main_key=None, **block)**
Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1  content1 ...
$SUBKEY2  content2 ...
```

which looks like the following:

FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)

**Parameters**

- **index** (int or None, optional) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

- **main_key** (string, optional) – Main keyword of the block that should be added (see: MKEYS). Default: the first main keyword of the file-type

- **block** (keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword.

If a
block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: `SUBKEY=content`

`add_content(self, content, main_index=None, sub_index=None, line_index=None)`
Add single-line content to the actual file.

**Parameters**

- `content (list)` – list containing one line of content given as a list of single statements
- `main_index (int, optional)` – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- `sub_index (int, optional)` – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- `line_index (int, optional)` – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

`add_copy_link(self, path, symlink=False)`
Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

**Parameters**

- `path (str)` – path to the existing file that should be copied
- `symlink (bool, optional)` – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

`add_main_keyword(self, key, main_index=None)`
Add a new main keyword (#key) to the actual file.

**Parameters**

- `key (string)` – key name
- `main_index (int, optional)` – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

`add_multi_content(self, content, main_index=None, sub_index=None)`
Add multiple content to the actual file.

**Parameters**

- `content (list)` – list containing lines of content, each given as a list of single statements
- `main_index (int, optional)` – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- `sub_index (int, optional)` – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
Notes
There needs to be at least one main keyword, otherwise the content is not added.
The content will be added at the end of the actual subkeyword.
If no sub keyword is present, a blank ("") will be added and the content is then directly connected
to the actual main keyword.

**add_sub_keyword** *(self, key, main_index=None, sub_index=None)*
Add a new sub keyword ($key) to the actual file.

**Parameters**
- **key** *(string)* – key name
- **main_index** *(int, optional)* – index of the corresponding main keyword
  where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – position, where the new sub keyword
  should be added between the existing ones. As default, it is placed at the end.

Notes
There needs to be at least one main keyword, otherwise the subkeyword is not added.

**append_to_block** *(self, index=None, **block)*
Append data to an existing Block in the actual file.
Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1 content1 ... $SUBKEY2 content2 ...
```
which looks like the following:

```
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**
- **index** *(int or None, optional)* – Positional index, where to insert the
  given Block. As default, it will be added at the end. Default: None.
- ****block**(keyword dict)* – here the dict-keywords are the ogs-subkeywords
  and the value is the content that should be added with this ogs-subkeyword
  If a block should contain content directly connected to a main keyword, use this main
  keyword as input-keyword and the content as value: SUBKEY=content

**check** *(self, verbose=True)*
Check if the given file is valid.

**Parameters**
- **verbose** *(bool, optional)* – Print information for the executed checks. Default: True

**Returns**
- **result** – Validity of the given file.
  **Return type** bool

**del_block** *(self, index=None, del_all=False)*
Delete a block by its index.

**Parameters**
• **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is returned. Default: None

• **del_all** (*bool, optional*) – State, if all blocks shall be deleted. Default: False

**del_content** (*self, main_index=-1, sub_index=-1, line_index=-1, del_all=False*)
Delete content by its position.

**Parameters**

• **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• **sub_index** (*int, optional*) – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.

• **line_index** (*int, optional*) – position of the content line, that should be deleted. Default: -1

• **del_all** (*bool, optional*) – State, if all content shall be deleted. Default: False

**del_copy_link** (*self*)
Remove a former given link to an external file.

**del_main_keyword** (*self, main_index=None, del_all=False*)
Delete a main keyword (#key) by its position.

**Parameters**

• **main_index** (*int, optional*) – position, which main keyword should be deleted. Default: -1

• **del_all** (*bool, optional*) – State, if all main keywords shall be deleted. Default: False

**del_sub_keyword** (*self, main_index=-1, sub_index=-1, del_all=False*)
Delete a sub keyword ($key) by its position.

**Parameters**

• **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• **pos** (*int, optional*) – position, which sub keyword should be deleted. Default: -1

• **del_all** (*bool, optional*) – State, if all sub keywords shall be deleted. Default: False

**get_block** (*self, index=None, as_dict=True*)
Get a Block from the actual file.

**Parameters**

• **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is returned. Default: None

• **as_dict** (*bool, optional*) – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for *add_block*. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no** (*self*)
Get the number of blocks in the file.

**get_file_type** (*self*)
Get the OGS file class name.
get_multi_keys(self, index=None)
State if a block has a unique set of sub keywords.

is_block_unique(self, index=None)
State if a block has a unique set of sub keywords.

read_file(self, path, encoding=None, verbose=False)
Read an existing OGS input file.

Parameters
• path (str) – path to the existing file that should be read
• encoding (str or None, optional) – encoding of the given file. If None is given, the system standard is used. Default: None
• verbose (bool, optional) – Print information of the reading process. Default: False

reset(self)
Delete every content.

save(self, path, **kwargs)
Save the actual OGS input file in the given path.

Parameters
• path (str) – path to where to file should be saved
• update (bool, optional) – state if the content should be updated before saving. Default: True

update_block(self, index=None, main_key=None, **block)
Update a Block from the actual file.

Parameters
• index (int or None, optional) – Positional index of the block of interest. As default, the last one is used. Default: None
• main_key (str, optional) – Main keyword of the block that should be updated (see: MKEYS). This shouldn’t be done. Default: None
• **block (keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

write_file(self)
Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['MEDIUM_PROPERTIES_DISTRIBUTED']
SKEYS = [['MSH_TYPE', 'MMP_TYPE', 'DIS_TYPE', 'CONVERSION_FACTOR', 'DATA']]
STD = {}

block_no
Number of blocks in the file.

file_name
base name of the file with extension.
Type str

file_path
save path of the file.
Type str
force_writing
state if the file is written even if empty.

Type bool

is_empty
State if the OGS file is empty.

name
name of the file without extension.

Type str

top_com
Top comment is ‘None’ for the MPD file.
ogs5py.fileclasses.msh

Class for the ogs MESH file.

Subpackages

The generators can be called with `MSH.generate`

```python
generator
Generators for the ogs MESH file.
```

File Class

```python
MSH([mesh_list])  
Class for a multi layer mesh file that contains multiple ‘#FEM_MSH’ Blocks.
```

```python
class MSH (mesh_list=None, **OGS_Config)
    Bases: ogs5py.fileclasses.msh.core.MSHsgl
    
    Class for a multi layer mesh file that contains multiple ‘#FEM_MSH’ Blocks.

    Parameters
    ----------
    * mesh_list (list of dict or None, optional) -- each dictionary contains one ‘#FEM_MSH’ block of the mesh file with with the following information (sorted by keys):
      mesh_data [dict] dictionary containing information about
      - AXISYMMETRY (bool)
      - CROSS_SECTION (bool)
      - PCS_TYPE (str)
      - GEO_TYPE (str)
      - GEO_NAME (str)
      - LAYER (int)
      nodes [ndarray] Array with all node positions
      elements [dict] contains nodelists for elements sorted by element types
      material_id [dict] contains material ids for each element sorted by element types
      element_id [dict] contains element ids for each element sorted by element types
    
    * task_root (str, optional) -- Path to the destiny model folder. Default: cwd+"ogs5model"
    
    * task_id (str, optional) -- Name for the ogs task. Default: “model”

    Attributes
    ----------
    AXISYMMETRY bool: AXISYMMETRY attribute.
    CROSS_SECTION bool: CROSS_SECTION attribute.
    ELEMENTS Get and set the ELEMENTS of the mesh.
    ELEMENT_ID Get and set the ELEMENT_IDS of the mesh.
```
ELEMENT_NO int: number of ELEMENTS.

ELEMENT_TYPES set: ELEMENT types of the mesh.

GEO_NAME str: GEO_NAME.

GEO_TYPE str: GEO_TYPE.

LAYER int: LAYER.

MATERIAL_ID Get and set the MATERIAL_IDs of the mesh.

MATERIAL_ID_flat Get flat version of the MATERIAL_IDs of the mesh.

NODES ndarray: (n,3) NODES of the mesh by its xyz-coordinates.

NODE_NO int: number of NODES.

PCS_TYPE str: PCS_TYPE.

block int: The actual block to access in the file.

block_no int: The number of blocks in the file.

center Get the mesh center.

centroids Get the centroids of the mesh.

centroids_flat Get flat version of the centroids of the mesh.

file_name str: base name of the file with extension.

file_path str: save path of the file.

force_writing bool: state if the file is written even if empty.

is_empty State if file is empty.

name str: name of the file without extension.

node_centroids Get the node centroids of the mesh.

node_centroids_flat Get flat version of the node centroids of the mesh.

volumes Get the volumes of the mesh-elements.

volumes_flat Get flat version of the volumes of the mesh-elements.

Methods

__call__ (self) Get a copy of the underlying dictionary.

add_copy_link(self, path[, symlink]) Add a link to copy a file instead of writing.

check(self[, verbose]) Check if the mesh is valid.

combine_mesh(self, ext_mesh, **kwargs) Combine this mesh with an external mesh.

del_copy_link(self) Remove a former given link to an external file.

export_mesh(self, filepath[, verbose]) Export the mesh to an unstructured mesh in different file-formats.

generate(self[, generator]) Use a mesh-generator from the generator module.

get_file_type(self) Get the OGS file class name.

import_mesh(self, filepath, **kwargs) Import an external unstructured mesh from different file-formats.

load(self, filepath, **kwargs) Load an OGS5 mesh from file.

read_file(self, path[, encoding, verbose]) Load an OGS5 mesh from file.

remove_dim(self, remove) Remove elements by given dimensions from a mesh.

reset(self) Delete every content.

Continued on next page
Table 53 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rotate(self, angle[, rotation_axis, ...])</code></td>
<td>Rotate a given mesh around a given rotation axis with a given angle.</td>
</tr>
<tr>
<td><code>save(self, path, \*\*kwargs)</code></td>
<td>Save the mesh to an OGS5 mesh file.</td>
</tr>
<tr>
<td><code>set_dict(self, mesh_dict)</code></td>
<td>Set an mesh as returned by tools methods or generators.</td>
</tr>
<tr>
<td><code>set_material_id(self[, material_id, ...])</code></td>
<td>Set material IDs by the corresponding element IDs.</td>
</tr>
<tr>
<td><code>shift(self, vector)</code></td>
<td>Shift a given mesh with a given vector.</td>
</tr>
<tr>
<td><code>show(self[, show_cell_data, ...])</code></td>
<td>Display the mesh colored by its material ID.</td>
</tr>
<tr>
<td><code>swap_axis(self[, axis1, axis2])</code></td>
<td>Swap axis of the coordinate system.</td>
</tr>
<tr>
<td><code>transform(self, xyz_func, \*\*kwargs)</code></td>
<td>Transform a given mesh with a given function “xyz_func”.</td>
</tr>
<tr>
<td><code>write_file(self)</code></td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>

__call__(self)

Get a copy of the underlying dictionary.

`add_copy_link(self, path, symlink=False)`

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

Parameters

- `path` *(str)* – path to the existing file that should be copied
- `symlink` *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

`check(self, verbose=True)`

Check if the mesh is valid.

Checked in the sense, that the contained data is consistent. Checks for correct element definitions or Node duplicates are not carried out.

Parameters `verbose` *(bool, optional)* – Print information for the executed checks. Default: True

Returns `result` – Validity of the given mesh.

Return type `bool`

`combine_mesh(self, ext_mesh, **kwargs)`

Combine this mesh with an external mesh.

The node list will be updated to eliminate duplicates. Element intersections are not checked. kwargs will be forwarded to “tools.combine”

Parameters

- `ext_mesh` *(mesh, dict or file)* – This is the mesh that should be added to the existing one.
- `decimals` *(int, optional)* – Number of decimal places to round the nodes to (default: 3). This will not round the output, it is just for comparison of the node vectors.
- `fast` *(bool, optional)* – If fast is True, the vector comparison is executed by a decimal comparison. If fast is False, all pairwise distances are calculated. Default: False

`del_copy_link(self)`

Remove a former given link to an external file.
export_mesh (self, filepath, verbose=False, **kwargs)

Export the mesh to an unstructured mesh in different file-formats.

kwargs will be forwarded to “tools.export_mesh”

Parameters

- **filepath (string)** – path to the file to export
- **file_format (str, optional)** – Here you can specify the fileformat. If ‘None’ it will be determined by file extension. Default: None
- **export_material_id (bool, optional)** – Here you can specify if the material_id should be exported. Default: True
- **export_element_id (bool, optional)** – Here you can specify if the element_id should be exported. Default: True
- **cell_data_by_id (ndarray or dict, optional)** – Here you can specify additional element data sorted by their IDs. It can be a dictionary with data-name as key and the ndarray as value. Default: None
- **point_data (ndarray or dict, optional)** – Here you can specify additional point data sorted by their IDs. It can be a dictionary with data-name as key and the ndarray as value. Default: None
- **field_data (ndarray or dict, optional)** – Here you can specify additional field data of the mesh. It can be a dictionary with data-name as key and the ndarray as value. Default: None

Notes

This routine calls the ‘write’ function from the meshio package and converts the input (see here: https://github.com/nschloe/meshio)

generate (self, generator='rectangular', **kwargs)

Use a mesh-generator from the generator module.

See: ogs5py.fileclasses.msh.generator

Parameters

- **generator (str)** – set the generator from the generator module
- ****kwargs – kwargs will be forwarded to the generator in use

Notes

The following generators are available:

rectangular([dim, mesh_origin, element_no, ...]) Generate a rectangular grid in 2D or 3D.
radial([dim, mesh_origin, angles, rad, z_arr]) Generate a radial grid in 2D or 3D.
grid_adapter2D([out_dim, out_res, ...]) in_dim, Generate a grid adapter.
grid_adapter3D([out_dim, in_dim, z_dim, ...]) in_dim, Generate a grid adapter.
block_adapter3D([xy_dim, in_res]) z_dim, Generate a block adapter.
gmsh(geo_object[, import_dim]) Generate mesh from pygmsh Geometry instance, gmsh code or gmsh .geo file.
**get_file_type** *(self)*
Get the OGS file class name.

**import_mesh** *(self, filepath, **kwargs)*
Import an external unstructured mesh from different file-formats.
kwarg will be forwarded to “tools.import_mesh”

**Parameters**
- **filepath** *(string)* – path to the mesh file to import
- **file_format** *(str, optional)* – Here you can specify the file format. If ‘None’ it will be determined by file extension. Default: None
- **ignore_unknown** *(bool, optional)* – Unknown data in the file will be ignored. Default: False
- **import_dim** *(iterable of int, optional)* – State which elements should be imported by dimensionality. Can be used to sort out unneeded elements for example from gmsh. Default: (1, 2, 3)

**Notes**
This routine calls the ‘read’ function from the meshio package and converts the output (see here: https://github.com/nschloe/meshio) If there is any “vertex” (0D element) in the element data, it will be removed.

**load** *(self, filepath, **kwargs)*
Load an OGS5 mesh from file.
kwarg will be forwarded to “tools.load ogs5msh”

**Parameters**
- **filepath** *(string)* – path to the '*.msh' OGS5 mesh file to load
- **verbose** *(bool, optional)* – Print information of the reading process. Default: True
- **ignore_unknown** *(bool, optional)* – Unknown data in the file will be ignored. Default: False
- **max_node_no** *(int, optional)* – If you know the maximal node number per elements in the mesh file, you can optimise the reading a bit. By default the algorithm will assume hexahedrons as 'largest' elements in the mesh. Default: 8
- **encoding** *(str or None, optional)* – encoding of the given file. If None is given, the system standard is used. Default: None

**Notes**
The **AREA** keyword within the Nodes definition is NOT supported and will violate the read data if present.

**read_file** *(self, path, encoding=None, verbose=False)*
Load an OGS5 mesh from file.

**Parameters**
- **path** *(str)* – path to the '*.msh' OGS5 mesh file to load
- **encoding** *(str or None, optional)* – encoding of the given file. If None is given, the system standard is used. Default: None
• **verbose** *(bool, optional)* – Print information of the reading process. Default: True

**remove_dim** *(self, remove)*
Remove elements by given dimensions from a mesh.

**Parameters**

**remove** *(iterable of int or single int)* – State which elements should be removed by dimensionality (1, 2, 3).

**reset** *(self)*
Delete every content.

**rotate** *(self, angle, rotation_axis=(0.0, 0.0, 1.0), rotation_point=(0.0, 0.0, 0.0))*
Rotate a given mesh around a given rotation axis with a given angle.

**Parameters**

• **angle** *(float)* – rotation angle given in radial length

• **rotation_axis** *(array_like, optional)* – Array containing the vector for rotation axis. Default: (0,0,1)

• **rotation_point** *(array_like, optional)* – Vector of the rotation base point. Default:(0,0,0)

**save** *(self, path, **kwargs)*
Save the mesh to an OGS5 mesh file.

kwargs will be forwarded to “tools.save_ogs5msh”

**Parameters**

• **path** *(string)* – path to the ‘*.msh’ OGS5 mesh file to save

• **verbose** *(bool, optional)* – Print information of the writing process. Default: True

**set_dict** *(self, mesh_dict)*
Set an mesh as returned by tools methods or generators.

Mesh will be checked for validity.

**Parameters**

**mesh_dict** *(dict or None, optional)* – Contains one ‘#FEM_MSH’ block of an OGS5 mesh file with the following information (sorted by keys):

  mesh_data [dict] dictionary containing information about
  • AXISYMMETRY (bool)
  • CROSS_SECTION (bool)
  • PCS_TYPE (str)
  • GEO_TYPE (str)
  • GEO_NAME (str)
  • LAYER (int)
  nodes [ndarray] Array with all node positions
  elements [dict] contains nodelists for elements sorted by element types
  material_id [dict] contains material ids for each element sorted by types
  element_id [dict] contains element ids for each element sorted by types

**set_material_id** *(self, material_id=0, element_id=None, element_mask=None)*
Set material IDs by the corresponding element IDs.

**Parameters**
• **material_id** *(int or ndarray, optional)* – The new material IDs. Either one value or an array. Default: 0

• **element_id** *(ndarray or None, optional)* – The corresponding element IDs, where to set the material IDs. If None, all elements are assumed and the material IDs are added by their index. Default: None

• **element_mask** *(ndarray or None, optional)* – Instead of the element IDs, one can specify a mask to select the element IDs. Default: None

**shift** *(self, vector)*
  Shift a given mesh with a given vector.

  **Parameters**
  
  • **vector** *(ndarray)* – array containing the shifting vector

**show** *(self, show_cell_data=None, show_material_id=False, show_element_id=False, log_scale=False)*
  Display the mesh colored by its material ID.

  **Parameters**

  • **show_cell_data** *(ndarray or dict, optional)* – Here you can specify additional element/cell data sorted by their IDs. It can be a dictionary with data-name as key and the ndarray as value. Default: None

  • **show_material_id** *(bool, optional)* – Here you can specify if the material_id should be shown. Default: False

  • **show_element_id** *(bool, optional)* – Here you can specify if the element_id should be shown. Default: False

  • **log_scale** *(bool, optional)* – State if the cell_data should be shown in log scale. Default: False

**Notes**
This routine needs “mayavi” to display the mesh. (see here: https://github.com/enthought/mayavi)

**swap_axis** *(self, axis1='y', axis2='z')*
  Swap axis of the coordinate system.

  **Parameters**

  • **axis1** *(str or int, optional)* – First selected Axis. Either in [“x”, “y”, “z”] or in [0, 1, 2]. Default: “y”

  • **axis2** *(str or int, optional)* – Second selected Axis. Either in [“x”, “y”, “z”] or in [0, 1, 2]. Default: “z”

**transform** *(self, xyz_func, **kwargs)*
  Transform a given mesh with a given function “xyz_func”.

  kwargs will be forwarded to “xyz_func”.

  **Parameters**

  • **xyz_func** *(function)* – the function transforming the points: x_new, y_new, z_new = f(x_old, y_old, z_old, **kwargs)

**write_file** *(self)*
  Write the actual OGS input file to the given folder.

  Its path is given by “task_root+task_id+file_ext”.

**AXISYMMETRY**
  AXISYMMETRY attribute.

  **Type** bool
CROSS SECTION
CROSS_SECTION attribute.

Type bool

ELEMENTS
Get and set the ELEMENTS of the mesh.

Notes
Type [dict of ndarrays] The elements are a dictionary sorted by their element-type
   “line” [ndarray of shape (n_line,2)] 1D element with 2 nodes
   “tri” [ndarray of shape (n_tri,3)] 2D element with 3 nodes
   “quad” [ndarray of shape (n_quad,4)] 2D element with 4 nodes
   “tet” [ndarray of shape (n_tet,4)] 3D element with 4 nodes
   “pyra” [ndarray of shape (n_pyra,5)] 3D element with 5 nodes
   “pris” [ndarray of shape (n_pris,6)] 3D element with 6 nodes
   “hex” [ndarray of shape (n_hex,8)] 3D element with 8 nodes

ELEMENT_ID
Get and set the ELEMENT_IDs of the mesh.
Standard element id order is given by:
   “line” “tri” “quad” “tet” “pyra” “pris” “hex”

Notes
Type [dict of ndarrays] The element IDs are a dictionary containing ints sorted by their element-type
   “line” [ndarray of shape (n_line,)] 1D element with 2 nodes
   “tri” [ndarray of shape (n_tri,)] 2D element with 3 nodes
   “quad” [ndarray of shape (n_quad,)] 2D element with 4 nodes
   “tet” [ndarray of shape (n_tet,)] 3D element with 4 nodes
   “pyra” [ndarray of shape (n_pyra,)] 3D element with 5 nodes
   “pris” [ndarray of shape (n_pris,)] 3D element with 6 nodes
   “hex” [ndarray of shape (n_hex,)] 3D element with 8 nodes

ELEMENT_NO
number of ELEMENTS.

Type int

ELEMENT_TYPES
ELEMENT types of the mesh.

Type set

GEO_NAME
GEO_NAME.

Type str

GEO_TYPE
GEO_TYPE.

Type str
LAYER
  LAYER.
  Type  int

MATERIAL_ID
  Get and set the MATERIAL_IDs of the mesh.

Notes
  Type  [dict of ndarrays] The material IDs are a dictionary containing ints sorted by their element-type
  “line”  [ndarray of shape (n_line,)] 1D element with 2 nodes
  “tri”  [ndarray of shape (n_tri,)] 2D element with 3 nodes
  “quad”  [ndarray of shape (n_quad,)] 2D element with 4 nodes
  “tet”  [ndarray of shape (n_tet,)] 3D element with 4 nodes
  “pyra”  [ndarray of shape (n_pyra,)] 3D element with 5 nodes
  “pris”  [ndarray of shape (n_pris,)] 3D element with 6 nodes
  “hex”  [ndarray of shape (n_hex,)] 3D element with 8 nodes

MATERIAL_ID_flat
  Get flat version of the MATERIAL_IDs of the mesh.

  See “mesh.MATERIAL_ID” This flattened MATERIAL_IDs are a stacked version of MATERIAL_ID, to get one continuous array. They are stacked in order of the ELEMENT_IDs. Standard stack order is given by:
  “line” “tri” “quad” “tet” “pyra” “pris” “hex”

Notes
  Type  [ndarray] The centroids are a list containing xyz-coordinates

NODES
  (n,3) NODES of the mesh by its xyz-coordinates.
  Type  ndarray

NODE_NO
  number of NODES.
  Type  int

PCS_TYPE
  PCS_TYPE.
  Type  str

block
  The actual block to access in the file.
  Type  int

block_no
  The number of blocks in the file.
  Type  int

center
  Get the mesh center.
**centroids**
Get the centroids of the mesh.

**Notes**

**Type** [dict of ndarrays] The centroids are a dictionary containing xyz-coordinates sorted by their element-type

- **“line”** [ndarray of shape (n_line,3)] 1D element with 2 nodes
- **“tri”** [ndarray of shape (n_tri,3)] 2D element with 3 nodes
- **“quad”** [ndarray of shape (n_quad,3)] 2D element with 4 nodes
- **“tet”** [ndarray of shape (n_tet,3)] 3D element with 4 nodes
- **“pyra”** [ndarray of shape (n_pyra,3)] 3D element with 5 nodes
- **“pris”** [ndarray of shape (n_pris,3)] 3D element with 6 nodes
- **“hex”** [ndarray of shape (n_hex,3)] 3D element with 8 nodes

**centroids_flat**
Get flat version of the centroids of the mesh.

See the “mesh.get_centroids” method. This flattened centroids are a stacked version of centroids, to get one continuous array. They are stacked in order of the element ids. Standard stack order is given by:

- **“line”**
- **“tri”**
- **“quad”**
- **“tet”**
- **“pyra”**
- **“pris”**
- **“hex”**

**Notes**

**Type** [ndarray] The centroids are a list containing xyz-coordinates

**file_name**
base name of the file with extension.

**Type** str

**file_path**
save path of the file.

**Type** str

**force_writing**
state if the file is written even if empty.

**Type** bool

**is_empty**
State if file is empty.

**name**
name of the file without extension.

**Type** str

**node_centroids**
Get the node centroids of the mesh.

**Notes**

**Type** [dict of ndarrays] The centroids are a dictionary containing xyz-coordinates sorted by their element-type

- **“line”** [ndarray of shape (n_line,3)] 1D element with 2 nodes
“tri” [ndarray of shape (n_tri,3)] 2D element with 3 nodes
“quad” [ndarray of shape (n_quad,3)] 2D element with 4 nodes
“tet” [ndarray of shape (n_tet,3)] 3D element with 4 nodes
“pyra” [ndarray of shape (n_pyra,3)] 3D element with 5 nodes
“pris” [ndarray of shape (n_pris,3)] 3D element with 6 nodes
“hex” [ndarray of shape (n_hex,3)] 3D element with 8 nodes

node_centroids_flat
Get flat version of the node centroids of the mesh.
See the “mesh.get_centroids” method. This flattened centroids are a stacked version of centroids, to get one continuous array. They are stacked in order of the element ids. Standard stack order is given by:

“line” “tri” “quad” “tet” “pyra” “pris” “hex”

Notes
Type [ndarray] The centroids are a list containing xyz-coordinates

volumes
Get the volumes of the mesh-elements.

Notes
Type [dict of ndarrays] The volumes are a dictionary containing the n-dimension volumes sorted by their element-type

“line” [ndarray of shape (n_line,3)] 1D element with 2 nodes
“tri” [ndarray of shape (n_tri,3)] 2D element with 3 nodes
“quad” [ndarray of shape (n_quad,3)] 2D element with 4 nodes
“tet” [ndarray of shape (n_tet,3)] 3D element with 4 nodes
“pyra” [ndarray of shape (n_pyra,3)] 3D element with 5 nodes
“pris” [ndarray of shape (n_pris,3)] 3D element with 6 nodes
“hex” [ndarray of shape (n_hex,3)] 3D element with 8 nodes

volumes_flat
Get flat version of the volumes of the mesh-elements.
This flattened volumes are a stacked version of centroids, to get one continuous array. They are stacked in order of the element ids. Standard stack order is given by:

“line” “tri” “quad” “tet” “pyra” “pris” “hex”

Notes
Type [ndarray] The volumes are a list containing the n-dimensional element volume
**ogsSpy.fileclasses.msh.generator**

Generators for the ogs MESH file.

### Generators

These generators can be called with `MSH.generate`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rectangular([dim, mesh_origin, element_no, ...])</code></td>
<td>Generate a rectangular grid in 2D or 3D.</td>
</tr>
<tr>
<td><code>radial([dim, mesh_origin, angles, rad, z_arr])</code></td>
<td>Generate a radial grid in 2D or 3D.</td>
</tr>
<tr>
<td><code>grid_adapter2D([out_dim, in_dim, out_res, ...])</code></td>
<td>Generate a grid adapter.</td>
</tr>
<tr>
<td><code>grid_adapter3D([out_dim, in_dim, z_dim, ...])</code></td>
<td>Generate a grid adapter.</td>
</tr>
<tr>
<td><code>block_adapter3D(xy_dim=10.0, z_dim=5.0, in_res=1.0)</code></td>
<td>Generate a block adapter.</td>
</tr>
<tr>
<td><code>gmsh(geo_object[, import_dim])</code></td>
<td>Generate mesh from pygmsh Geometry instance, gmsh code or gmsh .geo file.</td>
</tr>
</tbody>
</table>

**block_adapter3D** (xy_dim=10.0, z_dim=5.0, in_res=1.0)

Generate a block adapter.

It has a given resolution at the southern side with gmsh.

**Parameters**

- `xy_dim` *(float)* – xy-Dimension of the whole block
- `z_dim` *(float)* – z-Dimension of the whole block
- `in_res` *(float)* – Grid resolution at the southern side of the block

**Returns**

- `result` – Result contains one ‘#FEM_MSH’ block of the OGS mesh file with the following information (sorted by keys):
  - `mesh_data` [dict] dictionary containing information about
    - AXISYMMETRY (bool)
    - CROSS_SECTION (bool)
    - PCS_TYPE (str)
    - GEO_TYPE (str)
    - GEO_NAME (str)
    - LAYER (int)
  - `nodes` [ndarray] Array with all node postions
  - `elements` [dict] contains nodelists for elements sorted by element types
  - `material_id` [dict] contains material ids for each element sorted by element types
  - `element_id` [dict] contains element ids for each element sorted by element types

**Return type** dictionary

**gmsh(geo_object, import_dim=(1, 2, 3))**

Generate mesh from pygmsh Geometry instance, gmsh code or gmsh .geo file.

**Parameters**
ogs5py Documentation, Release 1.1.1

- **geo_object** *(str or list of str or Geometry instance from pygmsh)* – Either path to the gmsh .geo file, list of codelines for a .geo file or a pygmsh Geometry instance from pygmsh.

- **import_dim** *(iterable of int or single int, optional)* – State which elements should be imported by dimensionality. Default: (1, 2, 3)

**Returns**

result – Result contains one `#FEM_MSH` block of the OGS mesh file with the following information (sorted by keys):

- **mesh_data** *(dict)* dictionary containing information about
  - AXISYMMETRY *(bool)*
  - CROSS_SECTION *(bool)*
  - PCS_TYPE *(str)*
  - GEO_TYPE *(str)*
  - GEO_NAME *(str)*
  - LAYER *(int)*

- **nodes** *(ndarray)* Array with all node positions

- **elements** *(dict)* contains nodelists for elements sorted by element types

- **material_id** *(dict)* contains material ids for each element sorted by element types

- **element_id** *(dict)* contains element ids for each element sorted by element types

**Return type** *(dictionary)*

grid_adapter2D *(out_dim=(100.0, 100.0), in_dim=(50.0, 50.0), out_res=(10.0, 10.0), in_res=(1.0, 1.0), out_pos=(0.0, 0.0), in_pos=(25.0, 25.0), z_pos=0.0, in_mat=0, out_mat=0, fill=False)*

Generate a grid adapter.

2D adapter from an outer grid resolution to an inner grid resolution with gmsh.

**Parameters**

- **out_dim** *(list of 2 float)* – xy-Dimension of the outer block

- **in_dim** *(list of 2 float)* – xy-Dimension of the inner block

- **out_res** *(list of 2 float)* – Grid resolution of the outer block

- **in_res** *(list of 2 float)* – Grid resolution of the inner block

- **out_pos** *(list of 2 float)* – xy-Position of the origin of the outer block

- **in_pos** *(list of 2 float)* – xy-Position of the origin of the inner block

- **z_pos** *(float)* – z-Position of the origin of the whole block

- **in_mat** *(integer)* – Material-ID of the inner block

- **out_mat** *(integer)* – Material-ID of the outer block

- **fill** *(bool, optional)* – State if the inner block should be filled with a rectangular mesh. Default: False.

**Returns**

result – Result contains one `#FEM_MSH` block of the OGS mesh file with the following information (sorted by keys):

- **mesh_data** *(dict)* dictionary containing information about
• AXISYMMETRY (bool)
• CROSS SECTION (bool)
• PCS_TYPE (str)
• GEO_TYPE (str)
• GEO_NAME (str)
• LAYER (int)

**nodes** [ndarray] Array with all node positions

**elements** [dict] contains nodelists for elements sorted by element types

**material_id** [dict] contains material ids for each element sorted by element types

**element_id** [dict] contains element ids for each element sorted by element types

**Return type** dictionary

```python
grid_adapter3D(out_dim=(100.0, 100.0), in_dim=(50.0, 50.0), z_dim=-10.0, out_res=(10.0, 10.0, 10.0), in_res=(5.0, 5.0, 5.0), out_pos=(0.0, 0.0), in_pos=(25.0, 25.0), z_pos=0.0, in_mat=0, out_mat=0, fill=False)
```

Generate a grid adapter.

3D adapter from an outer grid resolution to an inner grid resolution with gmsh.

**Parameters**

- **out_dim** (list of 2 float) – xy-Dimension of the outer block
- **in_dim** (list of 2 float) – xy-Dimension of the inner block
- **z_dim** (float) – z-Dimension of the whole block
- **out_res** (list of 3 float) – Grid resolution of the outer block
- **in_res** (list of 3 float) – Grid resolution of the inner block
- **out_pos** (list of 2 float) – xy-Position of the origin of the outer block
- **in_pos** (list of 2 float) – xy-Position of the origin of the inner block
- **z_dim** – z-Position of the origin of the whole block
- **in_mat** (integer) – Material-ID of the inner block
- **out_mat** (integer) – Material-ID of the outer block
- **fill** (bool, optional) – State if the inner block should be filled with a rectangular mesh. Default: False.

**Returns**

**result** – Result contains one ‘#FEM_MSH’ block of the OGS mesh file with the following information (sorted by keys):

**mesh_data** [dict] dictionary containing information about

- AXISYMMETRY (bool)
- CROSS SECTION (bool)
- PCS_TYPE (str)
- GEO_TYPE (str)
- GEO_NAME (str)
- LAYER (int)

**nodes** [ndarray] Array with all node positions
elements  [dict] contains nodelists for elements sorted by element types
material_id  [dict] contains material ids for each element sorted by element types

element_id  [dict] contains element ids for each element sorted by element types

Return type  dictionary

radial(dim=3, mesh_origin=(0.0, 0.0, 0.0), angles=16, rad=array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), z_arr=array([0, -1]))
Generate a radial grid in 2D or 3D.

Parameters

• dim(int, optional) – Dimension of the resulting mesh, either 2 or 3. Default: 3
• mesh_origin(list of float, optional) – Origin of the mesh Default: [0.0, 0.0, 0.0]
• angles(int, optional) – Number of elements in each direction. Default: [10, 10, 10]
• rad(array, optional) – array of radii to set in the mesh
• z_arr(array, optional) – array of z values to set the layers in the mesh (only needed for dim=3) needs to be sorted in negative z direction

Returns

result  – Result contains one ‘#FEM_MSH’ block of the OGS mesh file with the following information (sorted by keys):

mesh_data  [dict] dictionary containing information about

• AXISYMMETRY (bool)
• CROSS_SECTION (bool)
• PCS_TYPE (str)
• GEO_TYPE (str)
• GEO_NAME (str)
• LAYER (int)

nodes  [ndarray] Array with all node postions
elements  [dict] contains nodelists for elements sorted by element types
material_id  [dict] contains material ids for each element sorted by element types

element_id  [dict] contains element ids for each element sorted by element types

Return type  dictionary

rectangular(dim=2, mesh_origin=(0.0, 0.0, 0.0), element_no=(10, 10, 10), element_size=(1.0, 1.0, 1.0))
Generate a rectangular grid in 2D or 3D.

Parameters

• dim(int, optional) – Dimension of the resulting mesh, either 2 or 3. Default: 3
• mesh_origin(list of float, optional) – Origin of the mesh Default: [0.0, 0.0, 0.0]
• element_no(list of int, optional) – Number of elements in each direction. Default: [10, 10, 10]
• **element_size** *(list of float, optional)* – Size of an element in each direction. Default: `[1.0,1.0,1.0]`

**Returns**

**result** – Result contains one ‘#FEM_MSH’ block of the OGS mesh file with the following information (sorted by keys):

- **mesh_data** *(dict)* dictionary containing information about
  - AXISYMMETRY *(bool)*
  - CROSS_SECTION *(bool)*
  - PCS_TYPE *(str)*
  - GEO_TYPE *(str)*
  - GEO_NAME *(str)*
  - LAYER *(int)*

- **nodes** *(ndarray)* Array with all node positions

- **elements** *(dict)* contains nodelists for elements sorted by element types

- **material_id** *(dict)* contains material ids for each element sorted by element types

- **element_id** *(dict)* contains element ids for each element sorted by element types

**Return type** dictionary
ogs5py Documentation, Release 1.1.1

ogs5py.fileclasses.msp

Class for the ogs SOLID_PROPERTIES file.

File Class

```python
class MSP(**OGS_Config)  
# Class for the ogs SOLID_PROPERTIES file.

Parameters

• task_root (str, optional) – Path to the destiny model folder. Default: cwd+"ogs5model"
• task_id (str, optional) – Name for the ogs task. Default: “model”
```

Notes

Main-Keywords (#):

• SOLID_PROPERTIES

Sub-Keywords ($) per Main-Keyword:

• SOLID_PROPERTIES
  – BIOT_CONSTANT
  – CREEP
  – DENSITY
  – ELASTICITY
  – EXCAVATION
  – E_Function
  – GRAVITY_CONSTANT
  – MICRO_STRUCTURE_PLAS
  – NAME
  – NON_REACTIVE_FRACTION
  – PLASTICITY
  – REACTIVE_SYSTEM
  – SOLID_BULK_MODULUS
  – SPECIFIC_HEAT_SOURCE
  – STRESS_INTEGRATION_TOLERANCE
  – STRESS_UNIT
  – SWELLING_PRESSURE_TYPE
  – THERMAL
  – THRESHOLD_DEV_STR

Chapter 3. ogs5py API
Time dependent Young's Poisson

Standard block: None


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rf_msp_new.cpp#L65

See also:

add_block

Attributes

- block_no Number of blocks in the file.
- file_name str: base name of the file with extension.
- file_path str: save path of the file.
- force_writing bool: state if the file is written even if empty.
- is_empty State if the OGS file is empty.
- name str: name of the file without extension.

Methods

- add_block(self[, index, main_key]) Add a new Block to the actual file.
- add_content(self, content[, main_index, ...]) Add single-line content to the actual file.
- add_copy_link(self, path[, symlink]) Add a link to copy a file instead of writing.
- add_main_keyword(self, key[, main_index]) Add a new main keyword (#key) to the actual file.
- add_multi_content(self, content[, ...]) Add multiple content to the actual file.
- add_sub_keyword(self[, key[, main_index, ...]]) Add a new sub keyword ($key) to the actual file.
- append_to_block(self[, index]) Append data to an existing Block in the actual file.
- check(self[, verbose]) Check if the given file is valid.
- del_block(self[, index, del_all]) Delete a block by its index.
- del_content(self[, main_index, sub_index, ...]) Delete content by its position.
- del_copy_link(self) Remove a former given link to an external file.
- del_main_keyword(self[, main_index, del_all]) Delete a main keyword (#key) by its position.
- del_sub_keyword(self[, main_index, ...]) Delete a sub keyword ($key) by its position.
- get_block(self[, index, as_dict]) Get a Block from the actual file.
- get_block_no(self) Get the number of blocks in the file.
- get_file_type(self) Get the OGS file class name.
- get_multi_keys(self[, index]) State if a block has a unique set of sub keywords.
- is_block_unique(self[, index]) State if a block has a unique set of sub keywords.
- read_file(self, path[, encoding, verbose]) Read an existing OGS input file.
- reset(self) Delete every content.
- save(self, path, **kwargs) Save the actual OGS input file in the given path.
- update_block(self[, index, main_key]) Update a Block from the actual file.
- write_file(self) Write the actual OGS input file to the given folder.

add_block (self, index=None, main_key=None, **block)
Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

#MAIN_KEY
which looks like the following:

```python
FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- `index` *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- `main_key` *(string, optional)* – Main keyword of the block that should be added (see: MKEYS). Default: the first main keyword of the file-type
- `**block` *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**add_content** *(self, content, main_index=None, sub_index=None, line_index=None)*  
Add single-line content to the actual file.

**Parameters**

- `content` *(list)* – list containing one line of content given as a list of single statements
- `main_index` *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- `sub_index` *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- `line_index` *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**
There needs to be at least one main keyword, otherwise the content is not added.  
If no sub keyword is present, a blank one (“” will be added and the content is then directly connected to the actual main keyword.

**add_copy_link** *(self, path, symlink=False)*  
Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/link to the target folder.

**Parameters**

- `path` *(str)* – path to the existing file that should be copied
- `symlink` *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword** *(self, key, main_index=None)*  
Add a new main keyword (#key) to the actual file.

**Parameters**

- `key` *(string)* – key name
- `main_index` *(int, optional)* – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.
add_multi_content(self, content, main_index=None, sub_index=None)

Add multiple content to the actual file.

**Parameters**

- **content** *(list)* – list containing lines of content, each given as a list of single statements
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one (“”) will be added and the content is then directly connected to the actual main keyword.

add_sub_keyword(self, key, main_index=None, sub_index=None)

Add a new sub keyword ($key) to the actual file.

**Parameters**

- **key** *(string)* – key name
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the subkeyword is not added.

append_to_block(self, index=None, **block)

Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY

$SUBKEY1 content1...

$SUBKEY2 content2...
```

which looks like the following:

FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)

**Parameters**

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- ****block**(keyword dict)** – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content
check (self, verbose=True)
Check if the given file is valid.

Parameters verbose (bool, optional) – Print information for the executed checks. Default: True

Returns result – Validity of the given file.

Return type bool
del_block (self, index=None, del_all=False)
Delete a block by its index.

Parameters
• index (int or None, optional) – Positional index of the block of interest. As default, the last one is returned. Default: None
• del_all (bool, optional) – State, if all blocks shall be deleted. Default: False
del_content (self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)
Delete content by its position.

Parameters
• main_index (int, optional) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
• sub_index (int, optional) – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.
• line_index (int, optional) – position of the content line, that should be deleted. Default: -1
• del_all (bool, optional) – State, if all content shall be deleted. Default: False
del_copy_link (self)
Remove a former given link to an external file.
del_main_keyword (self, main_index=None, del_all=False)
Delete a main keyword (#key) by its position.

Parameters
• main_index (int, optional) – position, which main keyword should be deleted. Default: -1
• del_all (bool, optional) – State, if all main keywords shall be deleted. Default: False
del_sub_keyword (self, main_index=-1, sub_index=-1, del_all=False)
Delete a sub keyword ($key) by its position.

Parameters
• main_index (int, optional) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
• pos (int, optional) – position, which sub keyword should be deleted. Default: -1
• del_all (bool, optional) – State, if all sub keywords shall be deleted. Default: False
get_block (self, index=None, as_dict=True)
Get a Block from the actual file.
get_block_no(self)
Get the number of blocks in the file.

get_file_type(self)
Get the OGS file class name.

get_multi_keys(self, index=None)
State if a block has a unique set of sub keywords.

is_block_unique(self, index=None)
State if a block has a unique set of sub keywords.

read_file(self, path, encoding=None, verbose=False)
Read an existing OGS input file.

Parameters
• path (str) – path to the existing file that should be read
• encoding (str or None, optional) – encoding of the given file. If None is given, the system standard is used. Default: None
• verbose (bool, optional) – Print information of the reading process. Default: False

reset(self)
Delete every content.

save(self, path, **kwargs)
Save the actual OGS input file in the given path.

Parameters
• path (str) – path to where to file should be saved
• update (bool, optional) – state if the content should be updated before saving. Default: True

update_block(self, index=None, main_key=None, **block)
Update a Block from the actual file.

Parameters
• index (int or None, optional) – Positional index of the block of interest. As default, the last one is used. Default: None
• main_key (string, optional) – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None
• **block (keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

write_file(self)
Write the actual OGS input file to the given folder.

Its path is given by "task_root+task_id+file_ext".

MKEYS = ['SOLID_PROPERTIES']
SKEYS = [['NAME', 'SWELLING_PRESSURE_TYPE', 'DENSITY', 'THERMAL', 'ELASTICITY', 'EXCAVATION', ... 'NON_REACTIVE_FRACTION', 'SPECIFIC_HEAT_SOURCE', 'ENTHALPY_CORRECTION_REFERENCE_TEMPERATURE', 'MICRO_STRUCTURE_PLAS']
STD = {}

block_no
   Number of blocks in the file.

file_name
   base name of the file with extension.
      Type str

file_path
   save path of the file.
      Type str

force_writing
   state if the file is written even if empty.
      Type bool

is_empty
   State if the OGS file is empty.

name
   name of the file without extension.
      Type str
class NUM(**OGS_Config)

Base: ogs5py.fileclasses.base.BlockFile

Class for the ogs NUMERICS file.

Parameters

- **task_root**(str, optional) – Path to the destiny model folder. Default: cwd+"ogs5model"
- **task_id**(str, optional) – Name for the ogs task. Default: “model”

Notes

Main-Keywords (#):

- NUMERICS

Sub-Keywords ($) per Main-Keyword:

- NUMERICS
  - COUPLED_PROCESS
  - COUPLING_CONTROL
  - COUPLING_ITERATIONS
  - DYNAMIC_DAMPING
  - ELE_GAUSS_POINTS
  - ELE_MASS_LUMPING
  - ELE_SUPG
  - ELE_UPWINDING
  - EXTERNAL_SOLVER_OPTION
  - FEM_FCT
  - GRAVITY_PROFILE
  - LINEAR_SOLVER
  - LOCAL_PICARD
  - NON_LINEAR_ITERATION
  - NON_LINEAR_SOLVER
  - NON_LINEAR_UPDATE_VELOCITY
  - OVERALL_COUPLING
  - PCS_TYPE
  - PLASTICITY_TOLERANCE
– **RENUMBER**

**Standard block:**

- **PCS_TYPE** “GROUNDWATER_FLOW”
- **LINEAR_SOLVER** [2, 5, 1.0e-14, 1000, 1.0, 100, 4]


**See also:**

- `add_block`

**Attributes**

- **block_no**: Number of blocks in the file.
- **file_name**: str: base name of the file with extension.
- **file_path**: str: save path of the file.
- **force_writing**: bool: state if the file is written even if empty.
- **is_empty**: State if the OGS file is empty.
- **name**: str: name of the file without extension.

**Methods**

- `add_block(self, index=None, main_key=None)`
  Add a new Block to the actual file.
- `add_content(self, content[, main_index, ...])`
  Add single-line content to the actual file.
- `add_copy_link(self, path[, symlink])`
  Add a link to copy a file instead of writing.
- `add_main_keyword(self, key[, main_index])`
  Add a new main keyword (#key) to the actual file.
- `add_multi_content(self, content[, ...])`
  Add multiple content to the actual file.
- `add_sub_keyword(self[, main_index, ...])`
  Add a new sub keyword ($key) to the actual file.
- `append_to_block(self[, index])`
  Append data to an existing Block in the actual file.
- `check(self[, verbose])`
  Check if the given file is valid.
- `del_block(self[, index, del_all])`
  Delete a block by its index.
- `del_content(self[, main_index, sub_index, ...])`
  Delete content by its position.
- `del_copy_link(self)`
  Remove a former given link to an external file.
- `del_main_keyword(self[, main_index, del_all])`
  Delete a main keyword (#key) by its position.
- `del_sub_keyword(self[, main_index, ...])`
  Delete a sub keyword ($key) by its position.
- `get_block(self[, index, as_dict])`
  Get a Block from the actual file.
- `get_block_no(self)`
  Get the number of blocks in the file.
- `get_file_type(self)`
  Get the OGS file class name.
- `get_multi_keys(self[, index])`
  State if a block has a unique set of sub keywords.
- `is_block_unique(self[, main_index])`
  State if a block has a unique set of sub keywords.
- `read_file(self, path[, encoding, verbose])`
  Read an existing OGS input file.
- `reset(self)`
  Delete every content.
- `save(self, path, **kwargs)`
  Save the actual OGS input file in the given path.
- `update_block(self[, index, main_key])`
  Update a Block from the actual file.
- `write_file(self)`
  Write the actual OGS input file to the given folder.

```python
add_block(self, index=None, main_key=None, **block)
```

Add a new Block to the actual file.
Keywords are the sub keywords of the actual file type:

```plaintext
#MAIN_KEY

$SUBKEY1  content1 ...
$SUBKEY2  content2 ...
```

which looks like the following:

```python
FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)
```

### Parameters

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- **main_key** *(string, optional)* – Main keyword of the block that should be added (see: MKEYs) Default: the first main keyword of the file-type
- **block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

```python
add_content(self, content, main_index=None, sub_index=None, line_index=None)
```

Add single-line content to the actual file.

### Parameters

- **content** *(list)* – list containing one line of content given as a list of single statements
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- **line_index** *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

### Notes

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

```python
add_copy_link(self, path, symlink=False)
```

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/link ed to the target folder.

### Parameters

- **path** *(str)* – path to the existing file that should be copied
- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

```python
add_main_keyword(self, key, main_index=None)
```

Add a new main keyword (#key) to the actual file.

### Parameters

- **key** *(string)* – key name
• **main_index** *(int, optional)* – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

### add_multi_content *(self, content, main_index=None, sub_index=None)*

Add multiple content to the actual file.

**Parameters**

- **content** *(list)* – list containing lines of content, each given as a list of single statements
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

---

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

### add_sub_keyword *(self, key, main_index=None, sub_index=None)*

Add a new sub keyword *(key)* to the actual file.

**Parameters**

- **key** *(string)* – key name
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

---

**Notes**

There needs to be at least one main keyword, otherwise the subkeyword is not added.

### append_to_block *(self, index=None, **block)*

Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY

$SUBKEY1  content1 ...

$SUBKEY2  content2...
```

which looks like the following:

FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)

**Parameters**

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
**block** *(keyword, dict)*—here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: `SUBKEY=content`

`check(self, verbose=True)`

Check if the given file is valid.

**Parameters**

- `verbose` *(bool, optional)*—Print information for the executed checks. Default: True

**Returns**

- `result`—Validity of the given file.

**Return type**

`bool`

`del_block(self, index=None, del_all=False)`

Delete a block by its index.

**Parameters**

- `index` *(int or None, optional)*—Positional index of the block of interest. As default, the last one is returned. Default: None
- `del_all` *(bool, optional)*—State, if all blocks shall be deleted. Default: False

`del_content(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)`

Delete content by its position.

**Parameters**

- `main_index` *(int, optional)*—index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- `sub_index` *(int, optional)*—index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.
- `line_index` *(int, optional)*—position of the content line, that should be deleted. Default: -1
- `del_all` *(bool, optional)*—State, if all content shall be deleted. Default: False

`del_copy_link(self)`

Remove a former given link to an external file.

`del_main_keyword(self, main_index=None, del_all=False)`

Delete a main keyword (#key) by its position.

**Parameters**

- `main_index` *(int, optional)*—position, which main keyword should be deleted. Default: -1
- `del_all` *(bool, optional)*—State, if all main keywords shall be deleted. Default: False

`del_sub_keyword(self, main_index=-1, sub_index=-1, del_all=False)`

Delete a sub keyword ($key) by its position.

**Parameters**

- `main_index` *(int, optional)*—index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- `pos` *(int, optional)*—position, which sub keyword should be deleted. Default: -1
• **del_all**(bool, optional) – State, if all sub keywords shall be deleted. Default: False

**get_block**(self, index=None, as_dict=True)
Get a Block from the actual file.

Parameters

• **index**(int or None, optional) – Positional index of the block of interest. As default, the last one is returned. Default: None

• **as_dict**(bool, optional) – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for add_block. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no**(self)
Get the number of blocks in the file.

**get_file_type**(self)
Get the OGS file class name.

**get_multi_keys**(self, index=None)
State if a block has a unique set of sub keywords.

**is_block_unique**(self, index=None)
State if a block has a unique set of sub keywords.

**read_file**(self, path, encoding=None, verbose=False)
Read an existing OGS input file.

Parameters

• **path**(str) – path to the existing file that should be read

• **encoding**(str or None, optional) – encoding of the given file. If None is given, the system standard is used. Default: None

• **verbose**(bool, optional) – Print information of the reading process. Default: False

**reset**(self)
Delete every content.

**save**(self, path, **kwargs)
Save the actual OGS input file in the given path.

Parameters

• **path**(str) – path to where to file should be saved

• **update**(bool, optional) – state if the content should be updated before saving. Default: True

**update_block**(self, index=None, main_key=None, **block)
Update a Block from the actual file.

Parameters

• **index**(int or None, optional) – Positional index of the block of interest. As default, the last one is used. Default: None

• **main_key**(string, optional) – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None

• **block**(keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: `SUBKEY=content`
write_file(self)
    Write the actual OGS input file to the given folder.
    Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['NUMERICS']

SKEYS = [['PCS_TYPE', 'RENUMBER', 'PLASTICITY_TOLERANCE', 'NON_LINEAR_ITERATION', 'NON_LINEAR_SOLVER', 'LINEAR_SOLVER', ... 'FEM_FCT', 'NEWTON_DAMPING', 'ADDITIONAL_NEWTON_TOLERANCES', 'REACTION_SCALING', 'METHOD']

STD = {'LINEAR_SOLVER': [2, 5, 1e-14, 1000, 1.0, 100, 4], 'PCS_TYPE': 'GROUNDWATER_FLOW'}

block_no
    Number of blocks in the file.

file_name
    base name of the file with extension.
    
        Type str

file_path
    save path of the file.
    
        Type str

force_writing
    state if the file is written even if empty.
    
        Type bool

is_empty
    State if the OGS file is empty.

name
    name of the file without extension.
    
        Type str
ogs5py.fileclasses.out

Class for the ogs OUTPUT file.

File Class

```python
class OUT(**OGS_Config)  # Class for the ogs OUTPUT file.
```

**Parameters**

- `task_root (str, optional)` – Path to the destiny model folder. Default: `cwd+"ogs5model"`
- `task_id (str, optional)` – Name for the ogs task. Default: “model”

**Notes**

Main-Keywords (#):

- OUTPUT
- VERSION

Sub-Keywords ($) per Main-Keyword:

- OUTPUT
  - NOD_VALUES
  - PCON_VALUES
  - ELE_VALUES
  - RWPT_VALUES
  - GEO_TYPE
  - TIM_TYPE
  - DAT_TYPE
  - VARIABLESHARING
  - AMPLIFIER
  - PCS_TYPE
  - DIS_TYPE
  - MSH_TYPE
  - MMP_VALUES
  - MFP_VALUES
  - TECPLOT_ZONE_SHARE
  - TECPLOT_ELEMENT_OUTPUT_CELL_CENTERED
  - TECPLOT_ZONES_FOR_MG
- VERSION
Standard block:

- **NOD_VALUES** “HEAD”
- **GEO_TYPE** “DOMAIN”
- **DAT_TYPE** “PVD”
- **TIM_TYPE** [“STEPS”, 1]


Reading routines:

See also:

- **add_block**

Attributes

- **block_no** Number of blocks in the file.
- **file_name** str: base name of the file with extension.
- **file_path** str: save path of the file.
- **force_writing** bool: state if the file is written even if empty.
- **is_empty** State if the OGS file is empty.
- **name** str: name of the file without extension.

Methods

- **add_block**(self[, index, main_key]) Add a new Block to the actual file.
- **add_content**(self, content[, main_index, ...]) Add single-line content to the actual file.
- **add_copy_link**(self, path[, symlink]) Add a link to copy a file instead of writing.
- **add_main_keyword**(self, key[, main_index]) Add a new main keyword (#key) to the actual file.
- **add_multi_content**(self, content[, ...]) Add multiple content to the actual file.
- **add_sub_keyword**(self, key[, main_index, ...]) Add a new sub keyword ($key) to the actual file.
- **append_to_block**(self[, index]) Append data to an existing Block in the actual file.
- **check**(self[, verbose]) Check if the given file is valid.
- **del_block**(self[, index, del_all]) Delete a block by its index.
- **del_content**(self[, main_index, sub_index, ...]) Delete content by its position.
- **del_copy_link**(self) Remove a former given link to an external file.
- **del_main_keyword**(self[, main_index, del_all]) Delete a main keyword (#key) by its position.
- **del_sub_keyword**(self[, main_index, ...]) Delete a sub keyword ($key) by its position.
- **get_block**(self[, index, as_dict]) Get a Block from the actual file.
- **get_block_no**(self) Get the number of blocks in the file.
- **get_file_type**(self) Get the OGS file class name.
- **get_multi_keys**(self[, index]) State if a block has a unique set of sub keywords.
- **is_block_unique**(self[, index]) State if a block has a unique set of sub keywords.
- **read_file**(self, path[, encoding, verbose]) Read an existing OGS input file.
Table 61 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reset(self)</code></td>
<td>Delete every content.</td>
</tr>
<tr>
<td><code>save(self, path, \*\*kwargs)</code></td>
<td>Save the actual OGS input file in the given path.</td>
</tr>
<tr>
<td><code>update_block(self[, index, main_key])</code></td>
<td>Update a Block from the actual file.</td>
</tr>
<tr>
<td><code>write_file(self)</code></td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>

Function: `add_block (self, index=None, main_key=None, **block)`

Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```
FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)
```

Parameters

- `index` *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- `main_key` *(string, optional)* – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type
- `**block` *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

Function: `add_content (self, content, main_index=None, sub_index=None, line_index=None)`

Add single-line content to the actual file.

Parameters

- `content` *(list)* – list containing one line of content given as a list of single statements
- `main_index` *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- `sub_index` *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- `line_index` *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

Notes

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

Function: `add_copy_link (self, path, symlink=False)`

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/link to the target folder.

Parameters
• **path** *(str)* – path to the existing file that should be copied
• **symlink** *(bool, optional)* – on UNIX systems it is possible to use a sym-
bolic link to save time if the file is big. Default: False

**add_main_keyword** *(self, key, main_index=None)*
Add a new main keyword (#key) to the actual file.

**Parameters**

• **key** *(string)* – key name

• **main_index** *(int, optional)* – position, where the new main keyword
should be added between the existing ones. As default, it is placed at the end.

**add_multi_content** *(self, content, main_index=None, sub_index=None)*
Add multiple content to the actual file.

**Parameters**

• **content** *(list)* – list containing lines of content, each given as a list of single
statements

• **main_index** *(int, optional)* – index of the corresponding main keyword
where the sub keyword should be added. As default, the last main keyword is
taken.

• **sub_index** *(int, optional)* – index of the corresponding sub keyword
where the content should be added. As default, the last sub keyword is taken.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.
The content will be added at the end of the actual subkeyword.
If no sub keyword is present, a blank one (""”) will be added and the content is then directly connected
to the actual main keyword.

**add_sub_keyword** *(self, key, main_index=None, sub_index=None)*
Add a new sub keyword ($key) to the actual file.

**Parameters**

• **key** *(string)* – key name

• **main_index** *(int, optional)* – index of the corresponding main keyword
where the sub keyword should be added. As default, the last main keyword is
taken.

• **sub_index** *(int, optional)* – position, where the new sub keyword
should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the subkeyword is not added.

**append_to_block** *(self, index=None, **block)*
Append data to an existing Block in the actual file.
Keywords are the sub keywords of the actual file type:

#MAIN_KEY

$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
which looks like the following:

```python
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- `**index**` *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

- `**block**` *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**check** *(self, verbose=True)*

Check if the given file is valid.

**Parameters**

- `verbose` *(bool, optional)* – Print information for the executed checks. Default: True

**Returns**

- `result` – Validity of the given file.

**Return type**

`bool`

**del_block** *(self, index=None, del_all=False)*

Delete a block by its index.

**Parameters**

- `**index**` *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

- `**del_all**` *(bool, optional)* – State, if all blocks shall be deleted. Default: False

**del_content** *(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)*

Delete content by its position.

**Parameters**

- `**main_index**` *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

- `**sub_index**` *(int, optional)* – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.

- `**line_index**` *(int, optional)* – position of the content line, that should be deleted. Default: -1

- `**del_all**` *(bool, optional)* – State, if all content shall be deleted. Default: False

**del_copy_link** *(self)*

Remove a former given link to an external file.

**del_main_keyword** *(self, main_index=None, del_all=False)*

Delete a main keyword (#key) by its position.

**Parameters**

- `**main_index**` *(int, optional)* – position, which main keyword should be deleted. Default: -1

- `**del_all**` *(bool, optional)* – State, if all main keywords shall be deleted. Default: False

**del_sub_keyword** *(self, main_index=-1, sub_index=-1, del_all=False)*

Delete a sub keyword ($key) by its position.

**Parameters**
• **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• **pos** (*int, optional*) – position, which sub keyword should be deleted. Default: -1

• **del_all** (*bool, optional*) – State, if all sub keywords shall be deleted. Default: False

**get_block** (*self, index=None, as_dict=True*)
Get a Block from the actual file.

**Parameters**

• **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is returned. Default: None

• **as_dict** (*bool, optional*) – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for add_block. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no** (*self*)
Get the number of blocks in the file.

**get_file_type** (*self*)
Get the OGS file class name.

**get_multi_keys** (*self, index=None*)
State if a block has a unique set of sub keywords.

**is_block_unique** (*self, index=None*)
State if a block has a unique set of sub keywords.

**read_file** (*self, path, encoding=None, verbose=False*)
Read an existing OGS input file.

**Parameters**

• **path** (*str*) – path to the existing file that should be read

• **encoding** (*str or None, optional*) – encoding of the given file. If None is given, the system standard is used. Default: None

• **verbose** (*bool, optional*) – Print information of the reading process. Default: False

**reset** (*self*)
Delete every content.

**save** (*self, path, **kwargs*)
Save the actual OGS input file in the given path.

**Parameters**

• **path** (*str*) – path to where to file should be saved

• **update** (*bool, optional*) – state if the content should be updated before saving. Default: True

**update_block** (*self, index=None, main_key=None, **block*)
Update a Block from the actual file.

**Parameters**

• **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is used. Default: None

• **main_key** (*string, optional*) – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None
**block**(key*word dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

```python
write_file(self)
Write the actual OGS input file to the given folder.
Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['OUTPUT', 'VERSION']
SKEYS = [['NOD_VALUES', 'PCON_VALUES', 'ELE_VALUES', 'RWPT_VALUES', 'GEO_TYPE', 'TIM_TYPE', 'DAT_TYPE', 'MFP_VALUES', 'TECPLOT_ZONE_SHARE', 'TECPLOT_ELEMENT_OUTPUT_CELL_CENTERED', 'TECPLOT_ZONES_FOR_MG'], ['']
STD = {'DAT_TYPE': 'PVD', 'GEO_TYPE': 'DOMAIN', 'NOD_VALUES': 'HEAD', 'TIM_TYPE': ['']

block_no
Number of blocks in the file.

file_name
base name of the file with extension.

    Type str

file_path
save path of the file.

    Type str

force_writing
state if the file is written even if empty.

    Type bool

is_empty
State if the OGS file is empty.

name
name of the file without extension.

    Type str
**ogs5py.fileclasses.pcs**

Class for the ogs PROCESS file.

---

### File Class

```python
class PCS(**OGS_Config)
    Bases: ogs5py.fileclasses.base.BlockFile
```

Class for the ogs PROCESS file.

**Parameters**

- `task_root (str, optional)` – Path to the destiny model folder. Default: `cwd+"ogs5model"`
- `task_id (str, optional)` – Name for the ogs task. Default: “model”

---

**Notes**

**Main-Keywords (#):**

- PROCESS

**Sub-Keywords ($) per Main-Keyword:**

- PROCESS
  - APP_TYPE
  - BOUNDARY_CONDITION_OUTPUT
  - COUNT
  - CPL_TYPE
  - DEACTIVATED_SUBDOMAIN
  - DISSOLVED_CO2_INGAS_PCS_NAME
  - DISSOLVED_CO2_PCS_NAME
  - TEMPERATURE_UNIT
  - ELEMENT_MATRIX_OUTPUT
  - GEO_TYPE
  - MEDIUM_TYPE
  - MEMORY_TYPE
  - MSH_TYPE
  - NEGLECT_H_INI_EFFECT
  - NUM_TYPE
  - OutputMassOfGasInModel
  - PCS_TYPE
  - PHASE_TRANSITION
  - PRIMARY_VARIABLE

---

3.6. ogs5py.fileclasses 167
– PROCESSED_BC
– RELOAD
– SATURATION_SWITCH
– SAVE_ECLIPSE_DATA_FILES
– SIMULATOR
– SIMULATOR_MODEL_PATH
– SIMULATOR_PATH
– SIMULATOR_WELL_PATH
– ST_RHS
– TIME_CONTROLLED_EXCAVATION
– TIM_TYPE
– UPDATE_INI_STATE
– USE_PRECALCULATED_FILES
– USE_VELOCITIES_FOR_TRANSPORT

Standard block:

PCS_TYPE “GROUNDWATER_FLOW”

NUM_TYPE “NEW”


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rf_pcs.cpp#L1803

See also:

add_block

Attributes

block_no Number of blocks in the file.

file_name str: base name of the file with extension.

file_path str: save path of the file.

force_writing bool: state if the file is written even if empty.

is_empty State if the OGS file is empty.
	name str: name of the file without extension.

Methods

add_block(self[, index, main_key]) Add a new Block to the actual file.

add_content(self, content[, main_index, ...]) Add single-line content to the actual file.

add_copy_link(self, path[, symlink]) Add a link to copy a file instead of writing.

add_main_keyword(self, key[, main_index]) Add a new main keyword (#key) to the actual file.

add_multi_content(self, content[, ...]) Add multiple content to the actual file.

add_sub_keyword(self, key[, main_index, ...]) Add a new sub keyword ($key) to the actual file.

append_to_block(self[, index]) Append data to an existing Block in the actual file.

check(self[, verbose]) Check if the given file is valid.

Continued on next page
Table 63 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>del_block(self[, index, del_all])</code></td>
<td>Delete a block by its index.</td>
</tr>
<tr>
<td><code>del_content(self[, main_index, sub_index, ...])</code></td>
<td>Delete content by its position.</td>
</tr>
<tr>
<td><code>del_copy_link(self)</code></td>
<td>Remove a former given link to an external file.</td>
</tr>
<tr>
<td><code>del_main_keyword(self[, main_index, del_all])</code></td>
<td>Delete a main keyword (#key) by its position.</td>
</tr>
<tr>
<td><code>del_sub_keyword(self[, main_index, ...])</code></td>
<td>Delete a sub keyword ($key) by its position.</td>
</tr>
<tr>
<td><code>get_block(self[, index, as_dict])</code></td>
<td>Get a Block from the actual file.</td>
</tr>
<tr>
<td><code>get_block_no(self)</code></td>
<td>Get the number of blocks in the file.</td>
</tr>
<tr>
<td><code>get_file_type(self)</code></td>
<td>Get the OGS file class name.</td>
</tr>
<tr>
<td><code>get_multi_keys(self[, index])</code></td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td><code>is_block_unique(self[, index])</code></td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td><code>read_file(self, path[, encoding, verbose])</code></td>
<td>Read an existing OGS input file.</td>
</tr>
<tr>
<td><code>reset(self)</code></td>
<td>Delete every content.</td>
</tr>
<tr>
<td><code>save(self, path, **kwargs)</code></td>
<td>Save the actual OGS input file in the given path.</td>
</tr>
<tr>
<td><code>update_block(self[, index, main_key])</code></td>
<td>Update a Block from the actual file.</td>
</tr>
<tr>
<td><code>write_file(self)</code></td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>

### add_block

#### docstring

Add a new Block to the actual file.

**Usage**

```python
add_block(self, index=None, main_key=None, **block)
```

**Parameters**

- `index` *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- `main_key` *(string, optional)* – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type
- `**block` *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

### add_content

#### docstring

Add single-line content to the actual file.

**Usage**

```python
add_content(self, content, main_index=None, sub_index=None, line_index=None)
```

**Parameters**

- `content` *(list)* – list containing one line of content given as a list of single statements
- `main_index` *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- `sub_index` *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- `line_index` *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.
Notes
There needs to be at least one main keyword, otherwise the content is not added.
If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

**add_copy_link** *(self, path, symlink=False)*
Add a link to copy a file instead of writing.
Instead of writing a file, you can give a path to an existing file, that will be copied/link to the target folder.

Parameters

- **path** *(str)* – path to the existing file that should be copied
- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword** *(self, key, main_index=None)*
Add a new main keyword (#key) to the actual file.

Parameters

- **key** *(string)* – key name
- **main_index** *(int, optional)* – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

**add_multi_content** *(self, content, main_index=None, sub_index=None)*
Add multiple content to the actual file.

Parameters

- **content** *(list)* – list containing lines of content, each given as a list of single statements
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

Notes
There needs to be at least one main keyword, otherwise the content is not added.
The content will be added at the end of the actual subkeyword.
If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

**add_sub_keyword** *(self, key, main_index=None, sub_index=None)*
Add a new sub keyword ($key) to the actual file.

Parameters

- **key** *(string)* – key name
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.
Notes
There needs to be at least one main keyword, otherwise the subkeyword is not added.

**append_to_block** *(self, index=None, **block)*
Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```plaintext
#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```plaintext
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**
- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- ****block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**check** *(self, verbose=True)*
Check if the given file is valid.

**Parameters**
- **verbose** *(bool, optional)* – Print information for the executed checks. Default: True

**Returns**
- **result** – Validity of the given file.

**Return type** boolean

**del_block** *(self, index=None, del_all=False)*
Delete a block by its index.

**Parameters**
- **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None
- **del_all** *(bool, optional)* – State, if all blocks shall be deleted. Default: False

**del_content** *(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)*
Delete content by its position.

**Parameters**
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.
- **line_index** *(int, optional)* – position of the content line, that should be deleted. Default: -1
- **del_all** *(bool, optional)* – State, if all content shall be deleted. Default: False
del_copy_link(self)
    Remove a former given link to an external file.

del_main_keyword(self, main_index=None, del_all=False)
    Delete a main keyword (#key) by its position.
    Parameters
    • main_index (int, optional) – position, which main keyword should be deleted. Default: -1
    • del_all (bool, optional) – State, if all main keywords shall be deleted. Default: False

del_sub_keyword(self, main_index=-1, sub_index=-1, del_all=False)
    Delete a sub keyword ($key) by its position.
    Parameters
    • main_index (int, optional) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
    • pos (int, optional) – position, which sub keyword should be deleted. Default: -1
    • del_all (bool, optional) – State, if all sub keywords shall be deleted. Default: False

get_block(self, index=None, as_dict=True)
    Get a Block from the actual file.
    Parameters
    • index (int or None, optional) – Positional index of the block of interest. As default, the last one is returned. Default: None
    • as_dict (bool, optional) – Here you can state if you want the output as a dictionary, which can be used as key-word-arguments for add_block. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

get_block_no(self)
    Get the number of blocks in the file.

get_file_type(self)
    Get the OGS file class name.

get_multi_keys(self, index=None)
    State if a block has a unique set of sub keywords.

is_block_unique(self, index=None)
    State if a block has a unique set of sub keywords.

read_file(self, path, encoding=None, verbose=False)
    Read an existing OGS input file.
    Parameters
    • path (str) – path to the existing file that should be read
    • encoding (str or None, optional) – encoding of the given file. If None is given, the system standard is used. Default: None
    • verbose (bool, optional) – Print information of the reading process. Default: False

reset(self)
    Delete every content.
save (self, path, **kwargs)
Save the actual OGS input file in the given path.

Parameters

• path (str) – path to where to file should be saved
• update (bool, optional) – state if the content should be updated before saving. Default: True

update_block (self, index=None, main_key=None, **block)
Update a Block from the actual file.

Parameters

• index (int or None, optional) – Positional index of the block of interest. As default, the last one is used. Default: None
• main_key (string, optional) – Main keyword of the block that should be updated (see: MKEYS). This shouldn’t be done. Default: None
• **block (keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

write_file (self)
Write the actual OGS input file to the given folder. Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['PROCESS']
SKEYS = [['PCS_TYPE', 'NUM_TYPE', 'CPL_TYPE', 'TIM_TYPE', 'APP_TYPE', 'COUNT', 'PRIMARY_VARIABLE', 'TEMPERATURE_UNIT', ..., 'DISSOLVED_CO2_INGAS_PCS_NAME', 'TIME_CONTROLLED_EXCAVATION', 'NEGLECT_H_INI_EFFECT', 'UPDATE_INI_STATE', 'CONSTANT']
STD = {'NUM_TYPE': 'NEW', 'PCS_TYPE': 'GROUNDWATER_FLOW'}

block_no
Number of blocks in the file.

declared
base name of the file with extension.

   Type str

file_path
save path of the file.

   Type str

force_writing
state if the file is written even if empty.

   Type bool

is_empty
State if the OGS file is empty.

   Type bool

declared
name of the file without extension.

   Type str
ogs5py Documentation, Release 1.1.1

ogs5py.fileclasses.pct

Class for the ogs PARTICLE DEFINITION file for RANDOM_WALK.

File Class

**PCT**([data, s_flag, task_root, task_id])

Class for the ogs Particle file, if the PCS TYPE is RANDOM_WALK.

class **PCT**

Parameters

- **data** *(np.array or None)* – particle data. Default: None
- **s_flag** *(int, optional)* – 1 for same pseudo-random series, 0 for different pseudo-random series. Default: 1
- **task_root** *(str, optional)* – Path to the destiny model folder. Default: cwd+"ogs5model"
- **task_id** *(str, optional)* – Name for the ogs task. Default: “model”

Attributes

- **file_name** *(str)*: base name of the file with extension.
- **file_path** *(str)*: save path of the file.
- **force_writing** *(bool)*: state if the file is written even if empty.
- **is_empty** *(bool)*: State if the OGS file is empty.
- **name** *(str)*: name of the file without extension.

Methods

- **add_copy_link**(self, path[, symlink])
  Add a link to copy a file instead of writing.
- **check**(self[, verbose])
  Check if the external geometry definition is valid.
- **del_copy_link**(self)
  Remove a former given link to an external file.
- **get_file_type**(self)
  Get the OGS file class name.
- **read_file**(self, path, **kwargs)
  Write the actual OGS input file to the given folder.
- **reset**(self)
  Delete every content.
- **save**(self, path)
  Save the actual PCT external file in the given path.
- **write_file**(self)
  Write the actual OGS input file to the given folder.

**add_copy_link**(self, path, symlink=False)

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

Parameters

- **path** *(str)* – path to the existing file that should be copied
- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a sym-
  bolic link to save time if the file is big. Default: False
check (self, verbose=True)
Check if the external geometry definition is valid.
   In the sence, that the contained data is consistent.
   
   Parameters verbose (bool, optional) – Print information for the executed checks. Default: True
   
   Returns result – Validity of the given gli.
   
   Return type bool

del_copy_link (self)
Remove a former given link to an external file.

get_file_type (self)
Get the OGS file class name.

read_file (self, path, **kwargs)
Write the actual OGS input file to the given folder.
   Its path is given by “task_root+task_id+file_ext”.

reset (self)
Delete every content.

save (self, path)
Save the actual PCT external file in the given path.
   
   Parameters path (str) – path to where to file should be saved

write_file (self)
Write the actual OGS input file to the given folder.
   Its path is given by “task_root+task_id+file_ext”.

file_name
   base name of the file with extension.
   
   Type str

file_path
   save path of the file.
   
   Type str

force_writing
   state if the file is written even if empty.
   
   Type bool

is_empty
   State if the OGS file is empty.

name
   name of the file without extension.
   
   Type str
ogs5py Documentation, Release 1.1.1

ogs5py.fileclasses.pqc

Class for the ogs PHREEQC interface file.

File Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PQC</strong>(<strong>OGS_Config)</strong></td>
<td>Class for the ogs PHREEQC interface file.</td>
</tr>
<tr>
<td><strong>PQCdat</strong>(<strong>OGS_Config)</strong></td>
<td>Class for the ogs PHREEQC dat file.</td>
</tr>
</tbody>
</table>

```python
class PQC(**OGS_Config)**
Bases: ogs5py.fileclasses.base.LineFile

Class for the ogs PHREEQC interface file.

Parameters

- **task_root**(str, optional) – Path to the destiny model folder. Default: cwd+"ogs5model"
- **task_id**(str, optional) – Name for the ogs task. Default: “model”

Notes

This is just handled as a line-wise file. You can access the data by line with:

PQC.lines


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rf_react.cpp#L2136

Attributes

- **file_name** str: base name of the file with extension.
- **file_path** str: save path of the file.
- **force_writing** bool: state if the file is written even if empty.
- **is_empty** bool: state if the file is empty.
- **name** str: name of the file without extension.

Methods

- **add_copy_link**(self, path[, symlink]) Add a link to copy a file instead of writing.
- **check**(self[, verbose]) Check if the given text-file is valid.
- **del_copy_link**(self) Remove a former given link to an external file.
- **get_file_type**(self) Get the OGS file class name.
- **read_file**(self, path[, encoding, verbose]) Read an existing OGS input file.
- **reset**(self) Delete every content.
- **save**(self, path) Save the actual line-wise file in the given path.
- **write_file**(self) Write the actual OGS input file to the given folder.

**add_copy_link**(self, path[, symlink=False]) Add a link to copy a file instead of writing.
Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

**Parameters**

- **path (str)** – path to the existing file that should be copied
- **symlink (bool, optional)** – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**check (self, verbose=True)**

Check if the given text-file is valid.

**Parameters**

- **verbose (bool, optional)** – Print information for the executed checks. Default: True

**Returns**

- **result** – Validity of the given file.

**Return type**

bool

**del_copy_link (self)**

Remove a former given link to an external file.

**get_file_type (self)**

Get the OGS file class name.

**read_file (self, path, encoding=None, verbose=False)**

Read an existing OGS input file.

**Parameters**

- **path (str)** – path to the existing file that should be read
- **encoding (str or None, optional)** – encoding of the given file. If None is given, the system standard is used. Default: None
- **verbose (bool, optional)** – Print information of the reading process. Default: False

**reset (self)**

Delete every content.

**save (self, path)**

Save the actual line-wise file in the given path.

**Parameters**

- **path (str)** – path to where to file should be saved

**write_file (self)**

Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

**file_name**

base name of the file with extension.

**Type**

str

**file_path**

save path of the file.

**Type**

str

**force_writing**

state if the file is written even if empty.

**Type**

bool

**is_empty**

state if the file is empty.

**Type**

bool
name

name of the file without extension.

Type  str

class PQCdat(**OGS_Config**)

Bases: `ogs5py.fileclasses.base.LineFile`

Class for the ogs PHREEQC dat file.

Parameters

- **task_root** (str, optional) – Path to the destiny model folder. Default: cwd+"ogs5model"
- **task_id** (str, optional) – Name for the ogs task. Default: “model”

Notes

This is just handled as a line-wise file. You can access the data by line with:

PQCdat.lines


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rf_react.cpp#L2136

Attributes

- **file_name** str: base name of the file with extension.
- **file_path** str: save path of the file.
- **force_writing** bool: state if the file is written even if empty.
- **is_empty** bool: state if the file is empty.
- **name** str: name of the file without extension.

Methods

- **add_copy_link**(self, path[, symlink])  
  Add a link to copy a file instead of writing.
- **check**(self[, verbose])  
  Check if the given text-file is valid.
- **del_copy_link**(self)  
  Remove a former given link to an external file.
- **get_file_type**(self)  
  Get the OGS file class name.
- **read_file**(self, path[, encoding, verbose])  
  Read an existing OGS input file.
- **reset**(self)  
  Delete every content.
- **save**(self, path)  
  Save the actual line-wise file in the given path.
- **write_file**(self)  
  Write the actual OGS input file to the given folder.

**add_copy_link**(self, path, symlink=False)  

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linkcd to the target folder.

Parameters

- **path** (str) – path to the existing file that should be copied
- **symlink** (bool, optional) – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False
check (self, verbose=True)
Check if the given text-file is valid.

Parameters verbose (bool, optional) – Print information for the executed checks. Default: True

Returns result – Validity of the given file.

Return type bool
del_copy_link (self)
Remove a former given link to an external file.

get_file_type (self)
Get the OGS file class name.

read_file (self, path, encoding=None, verbose=False)
Read an existing OGS input file.

Parameters

• path (str) – path to the existing file that should be read
• encoding (str or None, optional) – encoding of the given file. If None is given, the system standard is used. Default: None
• verbose (bool, optional) – Print information of the reading process. Default: False

reset (self)
Delete every content.

save (self, path)
Save the actual line-wise file in the given path.

Parameters path (str) – path to where to file should be saved

write_file (self)
Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

file_name
base name of the file with extension.

Type str

file_path
save path of the file.

Type str

force_writing
state if the file is written even if empty.

Type bool

is_empty
state if the file is empty.

Type bool

name
name of the file without extension.

Type str
Class for the ogs REACTION_INTERFACE file.

### File Class

**REI(**

**OGS_Config**

) Class for the ogs REACTION_INTERFACE file.

```python
class REI(**OGS_Config):  Class for the ogs REACTION_INTERFACE file.

Parameters:

- `task_root` (*str*, *optional*) – Path to the destiny model folder. Default: `cwd+"ogs5model"`
- `task_id` (*str*, *optional*) – Name for the ogs task. Default: “model”
```

### Notes

Main-Keywords (#):

- REACTION_INTERFACE

Sub-Keywords ($) per Main-Keyword:

- REACTION_INTERFACE
  - ALL_PCS_DUMP
  - DISSOLVED_NEUTRAL_CO2_SPECIES_NAME
  - HEATPUMP_2DH_TO_2DV
  - INITIAL_CONDITION_OUTPUT
  - MOL_PER
  - PCS_RENAME_INIT
  - PCS_RENAME_POST
  - PCS_RENAME_PRE
  - POROSITY_RESTART
  - PRESSURE
  - P_VLE
  - RESIDUAL
  - SODIUM_SPECIES_NAME
  - SOLID_SPECIES_DUMP_MOLE
  - TEMPERATURE
  - UPDATE_INITIAL_SOLID_COMPOSITION
  - VLE
  - WATER_CONCENTRATION
  - WATER_SATURATION_LIMIT
- WATER_SPECIES_NAME

Standard block: None


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rf_react_int.cpp#L173

See also:

add_block

Attributes

- block_no Number of blocks in the file.
- file_name str: base name of the file with extension.
- file_path str: save path of the file.
- force_writing bool: state if the file is written even if empty.
- is_empty State if the OGS file is empty.
- name str: name of the file without extension.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_block(self[, index, main_key])</td>
<td>Add a new Block to the actual file.</td>
</tr>
<tr>
<td>add_content(self, content[, main_index, ...])</td>
<td>Add single-line content to the actual file.</td>
</tr>
<tr>
<td>add_copy_link(self[, path[, symlink]])</td>
<td>Add a link to copy a file instead of writing.</td>
</tr>
<tr>
<td>add_main_keyword(self[, key[, main_index]])</td>
<td>Add a new main keyword (#key) to the actual file.</td>
</tr>
<tr>
<td>add_multi_content(self, content[, ...])</td>
<td>Add multiple content to the actual file.</td>
</tr>
<tr>
<td>add_sub_keyword(self[, key[, main_index, ...]])</td>
<td>Add a new sub keyword ($key) to the actual file.</td>
</tr>
<tr>
<td>append_to_block(self[, index])</td>
<td>Append data to an existing Block in the actual file.</td>
</tr>
<tr>
<td>check(self[, verbose])</td>
<td>Check if the given file is valid.</td>
</tr>
<tr>
<td>del_block(self[, index, del_all])</td>
<td>Delete a block by its index.</td>
</tr>
<tr>
<td>del_content(self[, main_index, sub_index, ...])</td>
<td>Delete content by its position.</td>
</tr>
<tr>
<td>del_copy_link(self)</td>
<td>Remove a former given link to an external file.</td>
</tr>
<tr>
<td>del_main_keyword(self[, main_index, del_all])</td>
<td>Delete a main keyword (#key) by its position.</td>
</tr>
<tr>
<td>del_sub_keyword(self[, main_index, ...])</td>
<td>Delete a sub keyword ($key) by its position.</td>
</tr>
<tr>
<td>get_block(self[, index, as_dict])</td>
<td>Get a Block from the actual file.</td>
</tr>
<tr>
<td>get_block_no(self)</td>
<td>Get the number of blocks in the file.</td>
</tr>
<tr>
<td>get_file_type(self)</td>
<td>Get the OGS file class name.</td>
</tr>
<tr>
<td>get_multi_keys(self[, index])</td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td>is_block_unique(self[, index])</td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td>read_file(self[, path[, encoding, verbose]])</td>
<td>Read an existing OGS input file.</td>
</tr>
<tr>
<td>reset(self)</td>
<td>Delete every content.</td>
</tr>
<tr>
<td>save(self, path[, **kwargs])</td>
<td>Save the actual OGS input file in the given path.</td>
</tr>
<tr>
<td>update_block(self[, index, main_key])</td>
<td>Update a Block from the actual file.</td>
</tr>
<tr>
<td>write_file(self)</td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>

add_block (self, index=None, main_key=None, **block)

Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

#MAIN_KEY
which looks like the following:

```python
FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- **main_key** *(string, optional)* – Main keyword of the block that should be added (see: MKEYS). Default: the first main keyword of the file-type
- **block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**add_content** *(self, content, main_index=None, sub_index=None, line_index=None)*

Add single-line content to the actual file.

**Parameters**

- **content** *(list)* – list containing one line of content given as a list of single statements
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- **line_index** *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

**add_copy_link** *(self, path, symlink=False)*

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied-linked to the target folder.

**Parameters**

- **path** *(str)* – path to the existing file that should be copied
- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword** *(self, key, main_index=None)*

Add a new main keyword (#key) to the actual file.

**Parameters**

- **key** *(string)* – key name
- **main_index** *(int, optional)* – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.
add_multi_content(self, content, main_index=None, sub_index=None)

Add multiple content to the actual file.

**Parameters**

- **content** (*list*) – list containing lines of content, each given as a list of single statements
- **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** (*int, optional*) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

add_sub_keyword(self, key, main_index=None, sub_index=None)

Add a new sub keyword ($key) to the actual file.

**Parameters**

- **key** (*string*) – key name
- **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** (*int, optional*) – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the subkeyword is not added.

append_to_block(self, index=None, **block)

Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1 content1...
$SUBKEY2 content2...
```

which looks like the following:

```
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- **index** (*int or None, optional*) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- ****block (*keyword dict*) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: `SUBKEY=content`
check\ ((self, \ verbose=True)\nCheck if the given file is valid.

Parameters verbose (bool, optional) – Print information for the executed checks. Default: True

Returns result – Validity of the given file.

Return type bool
del_block\ (self, \ index=None, \ del_all=False)\nDelete a block by its index.

Parameters
- index (int or None, optional) – Positional index of the block of interest. As default, the last one is returned. Default: None
- del_all (bool, optional) – State, if all blocks shall be deleted. Default: False
del_content\ (self, \ main_index=-1, \ sub_index=-1, \ line_index=-1, \ del_all=False)\nDelete content by its position.

Parameters
- main_index (int, optional) – Index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- sub_index (int, optional) – Index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.
- line_index (int, optional) – Position of the content line, that should be deleted. Default: -1
- del_all (bool, optional) – State, if all content shall be deleted. Default: False
del_copy_link\ (self)\nRemove a former given link to an external file.
del_main_keyword\ (self, \ main_index=None, \ del_all=False)\nDelete a main keyword (#key) by its position.

Parameters
- main_index (int, optional) – Position, which main keyword should be deleted. Default: -1
- del_all (bool, optional) – State, if all main keywords shall be deleted. Default: False
del_sub_keyword\ (self, \ main_index=-1, \ sub_index=-1, \ del_all=False)\nDelete a sub keyword ($key) by its position.

Parameters
- main_index (int, optional) – Index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- pos (int, optional) – Position, which sub keyword should be deleted. Default: -1
- del_all (bool, optional) – State, if all sub keywords shall be deleted. Default: False
get_block\ (self, \ index=None, \ as_dict=True)\nGet a Block from the actual file.
Parameters

- **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is returned. Default: None

- **as_dict** (*bool, optional*) – Here you can state if you want the output as a dictionary, which can be used as key-word-arguments for `add_block`. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no** *(self)*
Get the number of blocks in the file.

**get_file_type** *(self)*
Get the OGS file class name.

**get_multi_keys** *(self, index=None)*
State if a block has a unique set of sub keywords.

**is_block_unique** *(self, index=None)*
State if a block has a unique set of sub keywords.

**read_file** *(self, path, encoding=None, verbose=False)*
Read an existing OGS input file.

Parameters

- **path** (*str*) – path to the existing file that should be read

- **encoding** (*str or None, optional*) – encoding of the given file. If None is given, the system standard is used. Default: None

- **verbose** (*bool, optional*) – Print information of the reading process. Default: False

**reset** *(self)*
Delete every content.

**save** *(self, path, **kwargs)*
Save the actual OGS input file in the given path.

Parameters

- **path** (*str*) – path to where to file should be saved

- **update** (*bool, optional*) – state if the content should be updated before saving. Default: True

**update_block** *(self, index=None, main_key=None, **block)*
Update a Block from the actual file.

Parameters

- **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is used. Default: None

- **main_key** (*string, optional*) – Main keyword of the block that should be updated (see: MKEYS). This shouldn’t be done. Default: None

- ****block** (*keyword dict*) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**write_file** *(self)*
Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

**MKEYS** = ['REACTION_INTERFACE']

**SKEYS** = ['MOL_PER', 'WATER_CONCENTRATION', 'WATER_SPECIES_NAME', 'DISSOLVED_NEUTRAL_CO2_SPECIES_NAME']
STD = {}

block_no
   Number of blocks in the file.

file_name
   base name of the file with extension.
       Type str

file_path
   save path of the file.
       Type str

force_writing
   state if the file is written even if empty.
       Type bool

is_empty
   State if the OGS file is empty.

name
   name of the file without extension.
       Type str
ogs5py.fileclasses.rfd

Class for the ogs USER DEFINED TIME CURVES file.

File Class

```
RFD(**OGS_Config) Class for the ogs USER DEFINED TIME CURVES
file.
```

class RFD(**OGS_Config)

Bases: ogs5py.fileclasses.base.BlockFile

Class for the ogs USER DEFINED TIME CURVES file.

Parameters

- **task_root**(str, optional) – Path to the destiny model folder. Default: cwd+”ogs5model”
- **task_id**(str, optional) – Name for the ogs task. Default: “model”

Notes

Main-Keywords (#):

- PROJECT
- CURVE
- CURVES

Sub-Keywords ($) per Main-Keyword: (no sub-keywords)

Standard block: None


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/files0.cpp#L370

See also:

*add_block*

Attributes

- `block_no` Number of blocks in the file.
- `file_name` str: base name of the file with extension.
- `file_path` str: save path of the file.
- `force_writing` bool: state if the file is written even if empty.
- `is_empty` State if the OGS file is empty.
- `name` str: name of the file without extension.

Methods

- `add_block(self[, index, main_key])` Add a new Block to the actual file.
- `add_content(self, content[, main_index, ...])` Add single-line content to the actual file.

Continued on next page
Table 72 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_copy_link(self, path[, symlink])</code></td>
<td>Add a link to copy a file instead of writing.</td>
</tr>
<tr>
<td><code>add_main_keyword(self, key[, main_index])</code></td>
<td>Add a new main keyword (#key) to the actual file.</td>
</tr>
<tr>
<td><code>add_multi_content(self, content[, ...])</code></td>
<td>Add multiple content to the actual file.</td>
</tr>
<tr>
<td><code>add_sub_keyword(self, key[, main_index[, ...]])</code></td>
<td>Add a new sub keyword ($key) to the actual file.</td>
</tr>
<tr>
<td><code>append_to_block(self[, index])</code></td>
<td>Append data to an existing Block in the actual file.</td>
</tr>
<tr>
<td><code>check(self[, verbose])</code></td>
<td>Check if the given file is valid.</td>
</tr>
<tr>
<td><code>del_block(self[, index, del_all])</code></td>
<td>Delete a block by its index.</td>
</tr>
<tr>
<td><code>del_content(self[, main_index, sub_index[, ...]])</code></td>
<td>Delete content by its position.</td>
</tr>
<tr>
<td><code>del_copy_link(self)</code></td>
<td>Remove a former given link to an external file.</td>
</tr>
<tr>
<td><code>del_main_keyword(self[, main_index[, del_all]])</code></td>
<td>Delete a main keyword (#key) by its position.</td>
</tr>
<tr>
<td><code>del_sub_keyword(self[, main_index[, ...]])</code></td>
<td>Delete a sub keyword ($key) by its position.</td>
</tr>
<tr>
<td><code>get_block(self[, index, as_dict])</code></td>
<td>Get a Block from the actual file.</td>
</tr>
<tr>
<td><code>get_file_type(self)</code></td>
<td>Get the OGS file class name.</td>
</tr>
<tr>
<td><code>get_multi_keys(self[, index])</code></td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td><code>is_block_unique(self[, index])</code></td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td><code>read_file(self, path[, encoding, verbose])</code></td>
<td>Read an existing OGS input file.</td>
</tr>
<tr>
<td><code>reset(self)</code></td>
<td>Delete every content.</td>
</tr>
<tr>
<td><code>save(self, path, \*\*kwargs)</code></td>
<td>Save the actual OGS input file in the given path.</td>
</tr>
<tr>
<td><code>update_block(self[, index, main_key])</code></td>
<td>Update a Block from the actual file.</td>
</tr>
<tr>
<td><code>write_file(self)</code></td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>

add_block (self, index=None, main_key=None, **block)

Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

#MAIN_KEY

$SUBKEY1  content1 ...
$SUBKEY2  content2 ...

which looks like the following:

FILE.add_block (SUBKEY1=content1, SUBKEY2=content2)

Parameters

- `index` (int or None, optional) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- `main_key` (string, optional) – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type
- `**block` (keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

add_content (self, content, main_index=None, sub_index=None, line_index=None)

Add single-line content to the actual file.

Parameters

- `content` (list) – list containing one line of content given as a list of single statements
• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

• **line_index** *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

---

**add_copy_link** *(self, path, symlink=False)*

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

**Parameters**

• **path** *(str)* – path to the existing file that should be copied

• **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

---

**add_main_keyword** *(self, key, main_index=None)*

Add a new main keyword (#key) to the actual file.

**Parameters**

• **key** *(string)* – key name

• **main_index** *(int, optional)* – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

---

**add_multi_content** *(self, content, main_index=None, sub_index=None)*

Add multiple content to the actual file.

**Parameters**

• **content** *(list)* – list containing lines of content, each given as a list of single statements

• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

---

**add_sub_keyword** *(self, key, main_index=None, sub_index=None)*

Add a new sub keyword ($key) to the actual file.
Parameters

- **key** *(string)* – key name
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

Notes

There needs to be at least one main keyword, otherwise the subkeyword is not added.

**append_to_block**( self, index=None, **block**)

Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1  content1 ...
$SUBKEY2  content2 ...
```

which looks like the following:

```
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

Parameters

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- ****block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**check**( self, verbose=True)

Check if the given file is valid.

Parameters **verbose** *(bool, optional)* – Print information for the executed checks. Default: True

Returns **result** – Validity of the given file.

Return type **bool**

**del_block**( self, index=None, del_all=False)

Delete a block by its index.

Parameters

- **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None
- **del_all** *(bool, optional)* – State, if all blocks shall be deleted. Default: False

**del_content**( self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)

Delete content by its position.

Parameters

- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
• **sub_index** (*int, optional*) – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.

• **line_index** (*int, optional*) – position of the content line, that should be deleted. Default: -1

• **del_all** (*bool, optional*) – State, if all content shall be deleted. Default: False

`del_copy_link(self)`
Remove a former given link to an external file.

`del_main_keyword(self, main_index=None, del_all=False)`
Delete a main keyword (#key) by its position.

**Parameters**

• **main_index** (*int, optional*) – position, which main keyword should be deleted. Default: -1

• **del_all** (*bool, optional*) – State, if all main keywords shall be deleted. Default: False

`del_sub_keyword(self, main_index=-1, sub_index=-1, del_all=False)`
Delete a sub keyword ($key) by its position.

**Parameters**

• **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• **pos** (*int, optional*) – position, which sub keyword should be deleted. Default: -1

• **del_all** (*bool, optional*) – State, if all sub keywords shall be deleted. Default: False

`get_block(self, index=None, as_dict=True)`
Get a Block from the actual file.

**Parameters**

• **index** (*int or None, optional*) – Positional index of the block of interest. As default, the last one is returned. Default: None

• **as_dict** (*bool, optional*) – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for `add_block`. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

`get_block_no(self)`
Get the number of blocks in the file.

`get_file_type(self)`
Get the OGS file class name.

`get_multi_keys(self, index=None)`
State if a block has a unique set of sub keywords.

`is_block_unique(self, index=None)`
State if a block has a unique set of sub keywords.

`read_file(self, path, encoding=None, verbose=False)`
Read an existing OGS input file.

**Parameters**

• **path** (*str*) – path to the existing file that should be read
• **encoding** (str or None, optional) – encoding of the given file. If None is given, the system standard is used. Default: None

• **verbose** (bool, optional) – Print information of the reading process. Default: False

reset(self)
Delete every content.

save(self, path, **kwargs)
Save the actual OGS input file in the given path.

Parameters

• **path** (str) – path to where to file should be saved

• **update** (bool, optional) – state if the content should be updated before saving. Default: True

update_block(self, index=None, main_key=None, **block)
Update a Block from the actual file.

Parameters

• **index** (int or None, optional) – Positional index of the block of interest. As default, the last one is used. Default: None

• **main_key** (string, optional) – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None

• **block** (keyword dict) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

write_file(self)
Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['PROJECT', 'CURVE', 'CURVES', 'RENUMBER', 'ITERATION_PROPERTIES_CONCENTRATION', 'REFERENCE_CONDITIONS', 'APRIORI_REFINE_ELEMENT']

SKEYS = [[''], [''], [''], [''], [''], [''], ['']]

STD = {}

block_no
Number of blocks in the file.

file_name
base name of the file with extension.

    Type str

file_path
save path of the file.

    Type str

force_writing
state if the file is written even if empty.

    Type bool

is_empty
State if the OGS file is empty.

name
name of the file without extension.

    Type str
ogs5py.fileclasses.st

Class for the ogs SOURCE_TERM file.

File Class

```python
ST(**OGS_Config) Class for the ogs SOURCE_TERM file.
```

class ST(**OGS_Config)
    Bases: ogs5py.fileclasses.base.BlockFile
    Class for the ogs SOURCE_TERM file.

    Parameters
    *
    - task_root (str, optional) – Path to the destiny model folder. Default:
cwd+”ogs5model”
    - task_id (str, optional) – Name for the ogs task. Default: “model”

    Notes
    Main-Keywords (#):
    *
    - SOURCE_TERM

    Sub-Keywords ($) per Main-Keyword:
    *
    - SOURCE_TERM
      - AIR_BREAKING
      - CHANNEL
      - COMP_NAME
      - CONSTRAINED
      - DISTRIBUTIVE_VOLUME_FLUX
      - EPSILON
      - DIS_TYPE
      - EXPLICIT_SURFACE_WATER_PRESSURE
      - FCT_TYPE
      - GEO_TYPE
      - MSH_TYPE
      - NEGLECT_SURFACE_WATER_PRESSURE
      - NODE_AVERAGING
      - PCS_TYPE
      - PRIMARY_VARIABLE
      - TIME_INTERPOLATION
      - TIM_TYPE

    Standard block:

    PCS_TYPE “GROUNDWATER_FLOW”
**PRIMARY_VARIABLE** "HEAD"

**GEO_TYPE** ["POINT", "WELL"]

**DIS_TYPE** ["CONSTANT_NEUMANN", -1.0e-03]


Reading routines: https://github.com/ufz/ogs5/blob/master/FEM/rf_st_new.cpp#L221

See also:

`add_block`

**Attributes**

- `block_no` Number of blocks in the file.
- `file_name` str: base name of the file with extension.
- `file_path` str: save path of the file.
- `force_writing` bool: state if the file is written even if empty.
- `is_empty` State if the OGS file is empty.
- `name` str: name of the file without extension.

**Methods**

- `add_block(self[, index, main_key])`: Add a new Block to the actual file.
- `add_content(self[, main_index, ...])`: Add single-line content to the actual file.
- `add_copy_link(self[, path[, symlink]])`: Add a link to copy a file instead of writing.
- `add_main_keyword(self[, key[, main_index]])`: Add a new main keyword (#key) to the actual file.
- `add_multi_content(self[, content[, ...]])`: Add multiple content to the actual file.
- `add_sub_keyword(self[, key[, main_index, ...]])`: Add a new sub keyword ($key) to the actual file.
- `append_to_block(self[, index])`: Append data to an existing Block in the actual file.
- `check(self[, verbose])`: Check if the given file is valid.
- `del_block(self[, index, del_all])`: Delete a block by its index.
- `del_content(self[, main_index, sub_index, ...])`: Delete content by its position.
- `del_copy_link(self)`: Remove a former given link to an external file.
- `del_main_keyword(self[, main_index, del_all])`: Delete a main keyword (#key) by its position.
- `del_sub_keyword(self[, main_index, ...])`: Delete a sub keyword ($key) by its position.
- `get_block(self[, index, as_dict])`: Get a Block from the actual file.
- `get_block_no(self)`: Get the number of blocks in the file.
- `get_file_type(self)`: Get the OGS file class name.
- `get_multi_keys(self[, index])`: State if a block has a unique set of sub keywords.
- `is_block_unique(self[, index])`: State if a block has a unique set of sub keywords.
- `read_file(self[, path[, encoding, verbose]])`: Read an existing OGS input file.
- `reset(self)`: Delete every content.
- `save(self[, path[, encoding, verbose]])`: Save the actual OGS input file in the given path.
- `update_block(self[, index, main_key])`: Update a Block from the actual file.
- `write_file(self)`: Write the actual OGS input file to the given folder.

```python
add_block (self, index=None, main_key=None, **block)
```

Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:
#MAIN_KEY

$SUBKEY1 content1 ...

$SUBKEY2 content2 ...

which looks like the following:

FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)

Parameters

- **index** *(int or None, optional)* – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- **main_key** *(string, optional)* – Main keyword of the block that should be added (see: MKEYS) Default: the first main keyword of the file-type
- **block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

add_content *(self, content, main_index=None, sub_index=None, line_index=None)*

Add single-line content to the actual file.

Parameters

- **content** *(list)* – list containing one line of content given as a list of single statements
- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- **line_index** *(int, optional)* – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

Notes

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

add_copy_link *(self, path, symlink=False)*

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

Parameters

- **path** *(str)* – path to the existing file that should be copied
- **symlink** *(bool, optional)* – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

add_main_keyword *(self, key, main_index=None)*

Add a new main keyword (#key) to the actual file.

Parameters

- **key** *(string)* – key name
• **main_index** (*int, optional*) – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

**add_multi_content** (*self, content, main_index=None, sub_index=None*)
Add multiple content to the actual file.

**Parameters**

- **content** (*list*) – list containing lines of content, each given as a list of single statements
- **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** (*int, optional*) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

**add_sub_keyword** (*self, key, main_index=None, sub_index=None*)
Add a new sub keyword ($key) to the actual file.

**Parameters**

- **key** (*string*) – key name
- **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** (*int, optional*) – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the subkeyword is not added.

**append_to_block** (*self, index=None, **block*)
Append data to an existing Block in the actual file.

Keywords are the sub keywords of the actual file type:

```plaintext
#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```plaintext
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

- **index** (*int or None, optional*) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
• **block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: `SUBKEY=content`

**check** *(self, verbose=True)*
Check if the given file is valid.

Parameters **verbose** *(bool, optional)* – Print information for the executed checks. Default: True

Returns **result** – Validity of the given file.

Return type **bool**

**del_block** *(self, index=None, del_all=False)*
Delete a block by its index.

Parameters

• **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None

• **del_all** *(bool, optional)* – State, if all blocks shall be deleted. Default: False

**del_content** *(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)*
Delete content by its position.

Parameters

• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.

• **line_index** *(int, optional)* – position of the content line, that should be deleted. Default: -1

• **del_all** *(bool, optional)* – State, if all content shall be deleted. Default: False

**del_copy_link** *(self)*
Remove a former given link to an external file.

**del_main_keyword** *(self, main_index=None, del_all=False)*
Delete a main keyword (#key) by its position.

Parameters

• **main_index** *(int, optional)* – position, which main keyword should be deleted. Default: -1

• **del_all** *(bool, optional)* – State, if all main keywords shall be deleted. Default: False

**del_sub_keyword** *(self, main_index=-1, sub_index=-1, del_all=False)*
Delete a sub keyword ($key) by its position.

Parameters

• **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.

• **pos** *(int, optional)* – position, which sub keyword should be deleted. Default: -1
• **del_all** *(bool, optional)* – State, if all sub keywords shall be deleted.
  Default: False

**get_block** *(self, index=None, as_dict=True)*
Get a Block from the actual file.

  **Parameters**
  
  • **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None
  
  • **as_dict** *(bool, optional)* – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for **add_block**. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no** *(self)*
Get the number of blocks in the file.

**get_file_type** *(self)*
Get the OGS file class name.

**get_multi_keys** *(self, index=None)*
State if a block has a unique set of sub keywords.

**is_block_unique** *(self, index=None)*
State if a block has a unique set of sub keywords.

**read_file** *(self, path, encoding=None, verbose=False)*
Read an existing OGS input file.

  **Parameters**
  
  • **path** *(str)* – path to the existing file that should be read
  
  • **encoding** *(str or None, optional)* – encoding of the given file. If None is given, the system standard is used. Default: None
  
  • **verbose** *(bool, optional)* – Print information of the reading process. Default: False

**reset** *(self)*
Delete every content.

**save** *(self, path, **kwargs)*
Save the actual OGS input file in the given path.

  **Parameters**
  
  • **path** *(str)* – path to where to file should be saved
  
  • **update** *(bool, optional)* – state if the content should be updated before saving. Default: True

**update_block** *(self, index=None, main_key=None, **block)*
Update a Block from the actual file.

  **Parameters**
  
  • **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is used. Default: None
  
  • **main_key** *(string, optional)* – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None
  
  • **block** *(keyword dict)* – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkey. If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content
```python
write_file(self)
    Write the actual OGS input file to the given folder.
    Its path is given by “task_root+task_id+file_ext”.

MKEYS = ['SOURCE_TERM']
SKEYS = [['PCS_TYPE', 'PRIMARY_VARIABLE', 'COMP_NAME', 'GEO_TYPE', 'EPSILON', 'DIS_TYPE', 'NODE_AVERAGING', ...
STD = {'DIS_TYPE': [['CONSTANT_NEUMANN', -0.001], 'GEO_TYPE': [['POINT', 'WELL']]],

block_no
    Number of blocks in the file.

file_name
    base name of the file with extension.

    Type str

file_path
    save path of the file.

    Type str

force_writing
    state if the file is written even if empty.

    Type bool

is_empty
    State if the OGS file is empty.

name
    name of the file without extension.

    Type str
```
**ogs5py.fileclasses.tim**

Class for the ogs TIME_STEPING file.

**File Class**

```python
class TIM(**OGS_Config)
    Bases: ogs5py.fileclasses.base.BlockFile

    Class for the ogs TIME_STEPING file.

    Parameters
    • task_root (**str**, optional) – Path to the destiny model folder. Default: cwd+"ogs5model"
    • task_id (**str**, optional) – Name for the ogs task. Default: “model”
```

**Notes**

Main-Keywords (#):

• TIME_STEPING

Sub-Keywords ($) per Main-Keyword:

• TIME_STEPING
  – CRITICAL_TIME
  – INDEPENDENT
  – PCS_TYPE
  – SUBSTEPS
  – TIME_CONTROL
  – TIME_END
  – TIME_FIXED_POINTS
  – TIME_SPLITS
  – TIME_START
  – TIME_STEPS
  – TIME_UNIT

**Standard block:**

```plaintext
    PCS_TYPE  "GROUNDWATER_FLOW"
    TIME_START  0
    TIME_END   1000
    TIME_STEPS [10, 100]
```


See also:

`add_block`

**Attributes**

- `block_no`: Number of blocks in the file.
- `file_name`: str: base name of the file with extension.
- `force_writing`: bool: state if the file is written even if empty.
- `is_empty`: State if the OGS file is empty.
- `name`: str: name of the file without extension.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_block</code></td>
<td>Add a new Block to the actual file.</td>
</tr>
<tr>
<td><code>add_content</code></td>
<td>Add single-line content to the actual file.</td>
</tr>
<tr>
<td><code>add_copy_link</code></td>
<td>Add a link to copy a file instead of writing.</td>
</tr>
<tr>
<td><code>add_main_keyword</code></td>
<td>Add a new main keyword (#key) to the actual file.</td>
</tr>
<tr>
<td><code>add_multi_content</code></td>
<td>Add multiple content to the actual file.</td>
</tr>
<tr>
<td><code>add_sub_keyword</code></td>
<td>Add a new sub keyword ($key) to the actual file.</td>
</tr>
<tr>
<td><code>append_to_block</code></td>
<td>Append data to an existing Block in the actual file.</td>
</tr>
<tr>
<td><code>check</code></td>
<td>Check if the given file is valid.</td>
</tr>
<tr>
<td><code>del_block</code></td>
<td>Delete a block by its index.</td>
</tr>
<tr>
<td><code>del_content</code></td>
<td>Delete content by its position.</td>
</tr>
<tr>
<td><code>del_copy_link</code></td>
<td>Remove a former given link to an external file.</td>
</tr>
<tr>
<td><code>del_main_keyword</code></td>
<td>Delete a main keyword (#key) by its position.</td>
</tr>
<tr>
<td><code>del_sub_keyword</code></td>
<td>Delete a sub keyword ($key) by its position.</td>
</tr>
<tr>
<td><code>get_block</code></td>
<td>Get a Block from the actual file.</td>
</tr>
<tr>
<td><code>get_block_no</code></td>
<td>Get the number of blocks in the file.</td>
</tr>
<tr>
<td><code>get_file_type</code></td>
<td>Get the OGS file class name.</td>
</tr>
<tr>
<td><code>get_multi_keys</code></td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td><code>is_block_unique</code></td>
<td>State if a block has a unique set of sub keywords.</td>
</tr>
<tr>
<td><code>read_file</code></td>
<td>Read an existing OGS input file.</td>
</tr>
<tr>
<td><code>reset</code></td>
<td>Delete every content.</td>
</tr>
<tr>
<td><code>save</code></td>
<td>Save the actual OGS input file in the given path.</td>
</tr>
<tr>
<td><code>update_block</code></td>
<td>Update a Block from the actual file.</td>
</tr>
<tr>
<td><code>write_file</code></td>
<td>Write the actual OGS input file to the given folder.</td>
</tr>
</tbody>
</table>

```python
add_block(self, index=None, main_key=None, **block)
```

Add a new Block to the actual file.

Keywords are the sub keywords of the actual file type:

```
#MAIN_KEY

$SUBKEY1 content1 . . .

$SUBKEY2 content2 . . .
```

which looks like the following:

```python
FILE.add_block(SUBKEY1=content1, SUBKEY2=content2)
```
Parameters

- **index** (*int* or *None*, *optional*) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.
- **main_key** (*string*, *optional*) – Main keyword of the block that should be added (see: **MKEYS**) Default: the first main keyword of the file-type
- ****block** (*keyword dict*) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: **SUBKEY=content**

**add_content** (*self, content, main_index=None, sub_index=None, line_index=None*)

Add single-line content to the actual file.

Parameters

- **content** (*list*) – list containing one line of content given as a list of single statements
- **main_index** (*int*, *optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.
- **sub_index** (*int*, *optional*) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.
- **line_index** (*int*, *optional*) – position, where the new line of content should be added between the existing ones. As default, it is placed at the end.

Notes

There needs to be at least one main keyword, otherwise the content is not added.

If no sub keyword is present, a blank one (""") will be added and the content is then directly connected to the actual main keyword.

**add_copy_link** (*self, path, symlink=False*)

Add a link to copy a file instead of writing.

Instead of writing a file, you can give a path to an existing file, that will be copied/linked to the target folder.

Parameters

- **path** (*str*) – path to the existing file that should be copied
- **symlink** (*bool*, *optional*) – on UNIX systems it is possible to use a symbolic link to save time if the file is big. Default: False

**add_main_keyword** (*self, key, main_index=None*)

Add a new main keyword (#key) to the actual file.

Parameters

- **key** (*string*) – key name
- **main_index** (*int*, *optional*) – position, where the new main keyword should be added between the existing ones. As default, it is placed at the end.

**add_multi_content** (*self, content, main_index=None, sub_index=None*)

Add multiple content to the actual file.

Parameters

- **content** (*list*) – list containing lines of content, each given as a list of single statements
• **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** (*int, optional*) – index of the corresponding sub keyword where the content should be added. As default, the last sub keyword is taken.

**Notes**

There needs to be at least one main keyword, otherwise the content is not added.

The content will be added at the end of the actual subkeyword.

If no sub keyword is present, a blank one ("") will be added and the content is then directly connected to the actual main keyword.

### add_sub_keyword

```
add_sub_keyword(self, key, main_index=None, sub_index=None)
```

Add a new sub keyword ($key) to the actual file.

**Parameters**

• **key** (*string*) – key name

• **main_index** (*int, optional*) – index of the corresponding main keyword where the sub keyword should be added. As default, the last main keyword is taken.

• **sub_index** (*int, optional*) – position, where the new sub keyword should be added between the existing ones. As default, it is placed at the end.

**Notes**

There needs to be at least one main keyword, otherwise the subkeyword is not added.

### append_to_block

```
append_to_block(self, index=None, **block)
```

Append data to an existing Block in the actual file.

Keywords are the subkeywords of the actual file type:

```
#MAIN_KEY
$SUBKEY1 content1 ...
$SUBKEY2 content2 ...
```

which looks like the following:

```
FILE.append_to_block(SUBKEY1=content1, SUBKEY2=content2)
```

**Parameters**

• **index** (*int or None, optional*) – Positional index, where to insert the given Block. As default, it will be added at the end. Default: None.

• **block** (*keyword dict*) – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

### check

```
check(self, verbose=True)
```

Check if the given file is valid.

**Parameters** **verbose** (*bool, optional*) – Print information for the executed checks. Default: True

**Returns** **result** – Validity of the given file.
Return type: `bool`

**del_block** *(self, index=None, del_all=False)*
Delete a block by its index.

**Parameters**

- **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None
- **del_all** *(bool, optional)* – State, if all blocks shall be deleted. Default: False

**del_content** *(self, main_index=-1, sub_index=-1, line_index=-1, del_all=False)*
Delete content by its position.

**Parameters**

- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- **sub_index** *(int, optional)* – index of the corresponding sub keyword where the content should be deleted. As default, the last sub keyword is taken.
- **line_index** *(int, optional)* – position of the content line, that should be deleted. Default: -1
- **del_all** *(bool, optional)* – State, if all content shall be deleted. Default: False

**del_copy_link** *(self)*
Remove a former given link to an external file.

**del_main_keyword** *(self, main_index=None, del_all=False)*
Delete a main keyword (#key) by its position.

**Parameters**

- **main_index** *(int, optional)* – position, which main keyword should be deleted. Default: -1
- **del_all** *(bool, optional)* – State, if all main keywords shall be deleted. Default: False

**del_sub_keyword** *(self, main_index=-1, sub_index=-1, del_all=False)*
Delete a sub keyword ($key) by its position.

**Parameters**

- **main_index** *(int, optional)* – index of the corresponding main keyword where the sub keyword should be deleted. As default, the last main keyword is taken.
- **pos** *(int, optional)* – position, which sub keyword should be deleted. Default: -1
- **del_all** *(bool, optional)* – State, if all sub keywords shall be deleted. Default: False

**get_block** *(self, index=None, as_dict=True)*
Get a Block from the actual file.

**Parameters**

- **index** *(int or None, optional)* – Positional index of the block of interest. As default, the last one is returned. Default: None
• `as_dict (bool, optional)` – Here you can state of you want the output as a dictionary, which can be used as key-word-arguments for `add_block`. If False, you get the main-key, a list of sub-keys and a list of content. Default: True

**get_block_no** *(self)*
Get the number of blocks in the file.

**get_file_type** *(self)*
Get the OGS file class name.

**get_multi_keys** *(self, index=None)*
State if a block has a unique set of sub keywords.

**is_block_unique** *(self, index=None)*
State if a block has a unique set of sub keywords.

**read_file** *(self, path, encoding=None, verbose=False)*
Read an existing OGS input file.

Parameters

• `path (str)` – path to the existing file that should be read

• `encoding (str or None, optional)` – encoding of the given file. If None is given, the system standard is used. Default: None

• `verbose (bool, optional)` – Print information of the reading process. Default: False

**reset** *(self)*
Delete every content.

**save** *(self, path, **kwargs)*
Save the actual OGS input file in the given path.

Parameters

• `path (str)` – path to where to file should be saved

• `update (bool, optional)` – state if the content should be updated before saving. Default: True

**update_block** *(self, index=None, main_key=None, **block)*
Update a Block from the actual file.

Parameters

• `index (int or None, optional)` – Positional index of the block of interest. As default, the last one is used. Default: None

• `main_key (string, optional)` – Main keyword of the block that should be updated (see: MKEYS) This shouldn’t be done. Default: None

• `**block (keyword dict)` – here the dict-keywords are the ogs-subkeywords and the value is the content that should be added with this ogs-subkeyword If a block should contain content directly connected to a main keyword, use this main keyword as input-keyword and the content as value: SUBKEY=content

**write_file** *(self)*
Write the actual OGS input file to the given folder.

Its path is given by “task_root+task_id+file_ext”.

**MKEYS** = ['TIME_STEPING']

**SKEYS** = [['PCS_TYPE', 'TIME_START', 'TIME_END', 'TIME_UNIT', 'INDEPENDENT', 'TIME_STEPS', 'TIME_SPLITS', 'CRITICAL_TIME', 'TIME_CONTROL']

**STD** = {'PCS_TYPE': 'GROUNDWATER_FLOW', 'TIME_END': 1000, 'TIME_START': 0, 'TIME_STEPS':

**block_no**
Number of blocks in the file.
file_name
   base name of the file with extension.
   
   Type str

file_path
   save path of the file.
   
   Type str

force_writing
   state if the file is written even if empty.
   
   Type bool

is_empty
   State if the OGS file is empty.

name
   name of the file without extension.
   
   Type str
3.7 ogs5py.reader

ogs5py subpackage providing reader for the ogs5 output.

**Reader**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readvtk</td>
<td>A general reader for OGS vtk output files.</td>
</tr>
<tr>
<td>readpvd</td>
<td>Read a paraview pvd file.</td>
</tr>
<tr>
<td>readtec_point</td>
<td>Collect TECPLLOT point output from OGS5.</td>
</tr>
<tr>
<td>readtec_polyline</td>
<td>Collect TECPLLOT polyline output from OGS5.</td>
</tr>
<tr>
<td>VTK_ERR</td>
<td>vtkStringOutputWindow - File Specific output window class</td>
</tr>
</tbody>
</table>

**readvtk** *(task_root='.', task_id=None, pcs='ALL', single_file=None)*

A general reader for OGS vtk output files.

give a dictionary containing their data

the Filename of the pvd is structured the following way: `{task_id}[_[PCS]]xxxx.vtk` thereby the “_[PCS]” is optional and just present if a PCS_TYPE is specified in the *.out file

**Parameters**

- **task_root** *(string, optional)* – string containing the path to the directory containing the ogs output Default : “.”
- **task_id** *(string, optional)* – string containing the file name of the ogs task without extension Default : None
- **pcs** *(string or None, optional)* – specify the PCS type that should be collected Possible values are:
  - None/”” (no PCS_TYPE specified in *.out)
  - ”NO_PCS”
  - ”GROUNDWATER_FLOW”
  - ”LIQUID_FLOW”
  - ”RICHARDS_FLOW”
  - ”AIR_FLOW”
  - ”MULTI_PHASE_FLOW”
  - ”PS_GLOBAL”
  - ”HEAT_TRANSPORT”
  - ”DEFORMATION”
  - ”MASS_TRANSPORT”
  - ”OVERLAND_FLOW”
  - ”FLUID_MOMENTUM”
  - ”RANDOM_WALK”

You can get a list with all known PCS-types by setting PCS=”ALL” Default : None

- **single_file** *(string or None, optional)* – If you want to read just a single file, you can set the path here. Default : None
**Returns** result – keys are the point names and the items are the data from the corresponding files if pcs="ALL", the output is a dictionary with the PCS-types as keys

**Return type** dict

**readpvd** *(task_root='.', task_id=None, pcs='ALL', single_file=None)*

Read a paraview pvf file.

Convert all concerned files to a dictionary containing their data.

the Filename of the pvf is structured the following way: {task_id}[_{PCS}].pvf thereby the "_{PCS}" is optional and just present if a PCS_TYPE is specified in the *.out file

**Parameters**

- **task_root** *(string, optional)* – string containing the path to the directory containing the ogs output Default : ".
- **task_id** *(string, optional)* – string containing the file name of the ogs task without extension Default : None
- **pcs** *(string or None, optional)* – specify the PCS type that should be collected Possible values are:
  - None/"" (no PCS_TYPE specified in *.out)
  - "NO_PCS"
  - "GROUNDWATER_FLOW"
  - "LIQUID_FLOW"
  - "RICHARDS_FLOW"
  - "AIR_FLOW"
  - "MULTI_PHASE_FLOW"
  - "PS_GLOBAL"
  - "HEAT_TRANSPORT"
  - "DEFORMATION"
  - "MASS_TRANSPORT"
  - "OVERLAND_FLOW"
  - "FLUID_MOMENTUM"
  - "RANDOM_WALK"

You can get a list with all known PCS-types by setting PCS="ALL" Default : None

- **single_file** *(string or None, optional)* – If you want to read just a single file, you can set the path here. Default : None

**Returns** result – keys are the point names and the items are the data from the corresponding files if pcs="ALL", the output is a dictionary with the PCS-types as keys

**Return type** dict

**readtec_point** *(task_root='.', task_id=None, pcs='ALL', single_file=None)*

Collect TECPLOT point output from OGS5.

the Filenames are structured the following way: {task_id}_time_[NAME][_{PCS+extra}.tec thereby the "_{PCS}" is optional and just present if a PCS_TYPE is specified in the *.out file the “extra” will not be recognized and will destroy the search-process

**Parameters**

- **task_root** *(string, optional)* – string containing the path to the directory containing the ogs output Default : ".

**Returns** result – keys are the point names and the items are the data from the corresponding files if pcs="ALL", the output is a dictionary with the PCS-types as keys

**Return type** dict
• **task_id** *(string, optional)* – string containing the file name of the ogs task without extension Default : None

• **pcs** *(string or None, optional)* – specify the PCS type that should be collected Possible values are:
  
  – None/”” (no PCS_TYPE specified in *.out)
  – ”NO_PCS”
  – ”GROUNDWATER_FLOW”
  – ”LIQUID_FLOW”
  – ”RICHARDS_FLOW”
  – ”AIR_FLOW”
  – ”MULTI_PHASE_FLOW”
  – ”PS_GLOBAL”
  – ”HEAT_TRANSPORT”
  – ”DEFORMATION”
  – ”MASS_TRANSPORT”
  – ”OVERLAND_FLOW”
  – ”FLUID_MOMENTUM”
  – ”RANDOM_WALK”

  You can get a list with all known PCS-types by setting PCS=’ALL’ Default : None

• **single_file** *(string or None, optional)* – If you want to read just a single file, you can set the path here. Default : None

  **Returns** result – keys are the point names and the items are the data from the corresponding files if pcs=’ALL’, the output is a dictionary with the PCS-types as keys

  **Return type** dict

**readtec_polyline**(task_root=’.’, task_id=None, pcs=’ALL’, single_file=None, trim=True)

Collect TECPLLOT polyline output from OGS5.

the Filenames are structured the following way: {task_id}_ply_[NAME]_{t[ply_id]}_[{PCS}].tec thereby the ”_{[PCS]}” is optional and just present if a PCS_TYPE is specified in the *.out file

**Parameters**

• **task_root** *(string, optional)* – string containing the path to the directory containing the ogs output Default : ”.”

• **task_id** *(string, optional)* – string containing the file name of the ogs task without extension Default : None

• **pcs** *(string or None, optional)* – specify the PCS type that should be collected Possible values are:
  
  – None/”” (no PCS_TYPE specified in *.out)
  – ”NO_PCS”
  – ”GROUNDWATER_FLOW”
  – ”LIQUID_FLOW”
  – ”RICHARDS_FLOW”
  – ”AIR_FLOW”
  – ”MULTI_PHASE_FLOW”
You can get a list with all known PCS-types by setting pcs="ALL" Default : None

- **single_file** *(string or None, optional)* – If you want to read just a single file, you can set the path here. Default : None
- **trim** *(Bool, optional)* – if the ply_ids are not continuous, there will be “None” values in the output list. If trim is “True” these values will be eliminated. If there is just one output for a polyline, the list will be eliminated and the output will be the single dict. Default : True

**Returns** result – keys are the Polyline names and the items are lists sorted by the ply_id (it is assumed, that the ply_ids are continuous, if not, the corresponding list entries are “None”) if pcs="ALL", the output is a dictionary with the PCS-types as keys

**Return type** dict
3.8 ogs5py.tools

ogs5py subpackage providing tools.

**Subpackages**

<table>
<thead>
<tr>
<th>Subpackage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tools</td>
<td>Tools for the ogs5py package.</td>
</tr>
<tr>
<td>script</td>
<td>Script generator for ogs5py.</td>
</tr>
<tr>
<td>download</td>
<td>Downloader for ogs5.</td>
</tr>
<tr>
<td>output</td>
<td>Tools for ogs5py output files (independent from VTK package).</td>
</tr>
<tr>
<td>vtk_viewer</td>
<td>Viewer for a vtk file.</td>
</tr>
<tr>
<td>types</td>
<td>Type definitions for ogs5.</td>
</tr>
</tbody>
</table>
ogs5py Documentation, Release 1.1.1

ogs5py.tools.tools

Tools for the ogs5py package.

Classes

**Output**(file_or_name[, print_log])  A class to duplicate an output stream to stdout.

File related

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>search_mkey(fin)</code></td>
<td>Search for the first main keyword in a given file-stream.</td>
</tr>
<tr>
<td><code>uncomment(line)</code></td>
<td>Remove OGS comments from a given line of an OGS file.</td>
</tr>
<tr>
<td><code>is_key(sline)</code></td>
<td>Check if the given splitted line is an OGS key.</td>
</tr>
<tr>
<td><code>is_mkey(sline)</code></td>
<td>Check if the given splitted line is a main key.</td>
</tr>
<tr>
<td><code>is_skey(sline)</code></td>
<td>Check if the given splitted line is a sub key.</td>
</tr>
<tr>
<td><code>get_key(sline)</code></td>
<td>Get the key of a splitted line if there is any, else return &quot;&quot;.</td>
</tr>
<tr>
<td><code>find_key_in_list(key, key_list)</code></td>
<td>Look for the right corresponding key in a list.</td>
</tr>
<tr>
<td><code>format_dict(dict_in)</code></td>
<td>Format the dictionary to use upper-case keys.</td>
</tr>
<tr>
<td><code>format_content(content)</code></td>
<td>Format the content to be added to a 2D linewise array.</td>
</tr>
<tr>
<td><code>format_content_line(content)</code></td>
<td>Format a line of content to be a list of values.</td>
</tr>
<tr>
<td><code>guess_type(string)</code></td>
<td>Guess the type of a value given as string and return it accordingly.</td>
</tr>
<tr>
<td><code>search_task_id(task_root[, search_ext])</code></td>
<td>Search for OGS model names in the given path.</td>
</tr>
<tr>
<td><code>split_file_path(path[, abs_path])</code></td>
<td>Decompose a path to a file.</td>
</tr>
<tr>
<td><code>is_str_array(array)</code></td>
<td>A routine to check if an array contains strings.</td>
</tr>
</tbody>
</table>

Geometric tools

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rotate_points(points, angle[, ...])</code></td>
<td>Rotate points around a given rotation point and axis with a given angle.</td>
</tr>
<tr>
<td><code>shift_points(points, vector)</code></td>
<td>Shift points with a given vector.</td>
</tr>
<tr>
<td><code>transform_points(points, xyz_func, \*kwargs)</code></td>
<td>Transform points with a given function &quot;xyz_func&quot;.</td>
</tr>
<tr>
<td><code>hull_deform(x_in, y_in, z_in[, niv_top, ...])</code></td>
<td>Providing a transformation function to deform a given mesh.</td>
</tr>
<tr>
<td><code>rotation_matrix(vector, angle)</code></td>
<td>Create a rotation matrix.</td>
</tr>
<tr>
<td><code>volume(typ, \*pnt)</code></td>
<td>Volume of a OGS5 MeshElement.</td>
</tr>
<tr>
<td><code>centroid(typ, \*pnt)</code></td>
<td>Centroid of a OGS5 MeshElement.</td>
</tr>
</tbody>
</table>

Array tools

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unique_rows(data[, decimals, fast])</code></td>
<td>Unique made row-data with respect to given precision.</td>
</tr>
<tr>
<td><code>replace(arr, inval, outval)</code></td>
<td>Replace values of 'arr'.</td>
</tr>
<tr>
<td><code>by_id(array[, ids])</code></td>
<td>Return a flattend array side-by-side with the array-element ids.</td>
</tr>
<tr>
<td><code>specialrange(val_min, val_max, steps[, typ])</code></td>
<td>Calculation of special point ranges.</td>
</tr>
<tr>
<td><code>generate_time(time_array[, time_start, ...])</code></td>
<td>Return a dictionary for the &quot;*.tim&quot; file.</td>
</tr>
</tbody>
</table>

212 Chapter 3. ogs5py API
class Output (file_or_name, print_log=True)
Bases: object

A class to duplicate an output stream to stdout.

Parameters

• file_or_name (filename or open filehandle (writable)) – File that will be duplicated

• print_log (bool, optional) – State if log should be printed. Default: True

Methods

close (self) Close the file and restore the channel.
flush (self) Flush both channels.
write (self, data) Write data to both channels.

by_id (array, ids=None)
Return a flattened array side-by-side with the array-element ids.

Parameters

• array (array-like) – Input data. will be flattened.

• ids (None or array-like) – You can provide specific ids if needed. As default, the array-ids are used. Default: None

Returns

Return type zipped (id, array) object

centroid (typ, *pnt)
Centroid of a OGS5 Meshelement.

Parameters

• typ (string) – OGS5 Meshelement type. Should be one of the following:
  – "line": 1D element with 2 nodes
  – "tri": 2D element with 3 nodes
  – "quad": 2D element with 4 nodes
  – "tet": 3D element with 4 nodes
  – "pyra": 3D element with 5 nodes
  – "pris": 3D element with 6 nodes
  – "hex": 3D element with 8 nodes

• *pnt (Node Coordinates pnt = (x_0, x_1, ...)) – List of points defining the Meshelement. A point is given as an (x,y,z) tuple and for each point, there can be a stack of points, if the volume should be calculated for multiple elements of the same type.

Returns centroid – Array containing the Centroids of the give elements.
Return type  ndarray

Notes
The calculation is performed by geometric decomposition of the elements.
https://en.wikipedia.org/wiki/Centroid#By_geometric_decomposition

**find_key_in_list** *(key, key_list)*
Look for the right corresponding key in a list.
key has to start with an given key from the list and the longest key will be returned.

Parameters
- **key** *(str)* – Given key.
- **key_list** *(list of str)* – Valid keys to be checked against.

Returns **found_key** – The best match. None if nothing was found.

Return type  str or None

**format_content** *(content)*
Format the content to be added to a 2D linewise array.

Parameters **content** *(anything)* – Single object, or list of objects, or list of lists of objects.

**format_content_line** *(content)*
Format a line of content to be a list of values.

Parameters **content** *(anything)* – Single object, or list of objects

**format_dict** *(dict_in)*
Format the dictionary to use upper-case keys.

Parameters **dict_in** *(dict)* – input dictionary

**generate_time** *(time_array, time_start=0, factors=1, is_diff=False)*
Return a dictionary for the ".tim" file.

Parameters
- **time_array** *(array-like)* – Input time. will be flattened. Either time step sizes for each step, (is_diff=True) or an array of time-points.
- **time_start** *(float, optional)* – Starting point for time stepping. Default: 0
- **factors** *(int or array-like, optional)* – Repeating factors for each time step. Default: 1
- **is_diff** *(bool, optional)* – State if the given time array contains only the step size for each step. Default: False

Returns **dict** – keys: {“TIME_START”, “TIME_END”, “TIME_STEPS”}

Return type  input dict for ".tim".

**get_key** *(sline)*
Get the key of a splitted line if there is any, else return “”.

Parameters **sline** *(list of str)* – given splitted line

**guess_type** *(string)*
Guess the type of a value given as string and return it accordingly.

Parameters **string** *(str)* – given string containing the value
**hull_deform**(*x_in, y_in, z_in, niv_top=10.0, niv_bot=0.0, func_top=None, func_bot=None, direction='z')

Providing a transformation function to deform a given mesh.

Transformation is in a given direction by a self defined hull-functions $z = \text{func}(x, y)$. Could be used with `transform_mesh` and `transform_points`.

**Parameters**

- **x_in** (*ndarray*) – Array of the x-positions
- **y_in** (*ndarray*) – Array of the y-positions
- **z_in** (*ndarray*) – Array of the z-positions
- **niv_top** (*float*) – height of the top niveau to be deformed by *func_top*
- **niv_bot** (*float*) – height of the bottom niveau to be deformed by *func_bot*
- **func_top** (*function or float*) – function deforming the top niveau: $z_{\text{top}} = \text{func}_\text{top}(x, y)$
- **func_bot** (*function or float*) – function deforming the bottom niveau: $z_{\text{bot}} = \text{func}_\text{bot}(x, y)$
- **direction** (*string, optional*) – defining the direction of deforming. This direction will be used as z-value. Default: “z”

**Returns**

- **x_out**, **y_out**, **z_out** – transformed arrays

**Return type** *ndarray*

**is_key**(*sline*)

Check if the given splitted line is an OGS key.

**Parameters**

- **sline** (*list of str*) – given splitted line

**is_mkey**(*sline*)

Check if the given splitted line is a main key.

**Parameters**

- **sline** (*list of str*) – given splitted line

**is_skey**(*sline*)

Check if the given splitted line is a sub key.

**Parameters**

- **sline** (*list of str*) – given splitted line

**is_str_array**(*array*)

A routine to check if an array contains strings.

**Parameters**

- **array** (*iterable*) – array to check

**Returns**

**Return type** *bool*

**replace**(*arr, inval, outval*)

Replace values of ‘arr’.

Replace values defined in ‘inval’ with values defined in ‘outval’.

**Parameters**

- **arr** (*ndarray*) – array containing the input data
- **inval** (*ndarray*) – values appearing in ‘arr’ that should be replaced
- **outval** (*ndarray*) – values that should be written in ‘arr’ instead of values in ‘inval’

**Returns**

- **result** – array of the same shape as ‘arr’ containing the new data

**Return type** *ndarray*
rotate_points(points, angle, rotation_axis=(0.0, 0.0, 1.0), rotation_point=(0.0, 0.0, 0.0))

Rotate points around a given rotation point and axis with a given angle.

Parameters

- **points** (ndarray) – Array with all points positions.
- **angle** (float) – Rotation angle given in radial length
- **rotation_axis** (array_like, optional) – Array containing the vector for rotation axis. Default: (0,0,1)
- **rotation_point** (array_like, optional) – Array containing the vector for rotation base point. Default: (0,0,0)

Returns **new_array** – rotated array

Return type **ndarray**

rotation_matrix(vector, angle)

Create a rotation matrix.

For rotation around a given vector with a given angle.

Parameters

- **vector** (ndarray) – array containing the vector for rotation axis
- **angle** (float) – rotation angle given in radial length

Returns **result** – matrix to be used for matrix multiplication with vectors to be rotated.

Return type **ndarray**

search_mkey(fin)

Search for the first main keyword in a given file-stream.

Parameters **fin** (stream) – given opened file

search_task_id(task_root, search_ext=None)

Search for OGS model names in the given path.

Parameters

- **task_root** (str) – Path to the destiny folder.
- **search_ext** (str) – OGS extension that should be searched for. Default: All known.

Returns **found_ids** – List of all found task_ids.

Return type **list of str**

shift_points(points, vector)

Shift points with a given vector.

Parameters

- **points** (ndarray) – Array with all points positions.
- **vector** (ndarray) – array containing the shifting vector

Returns **new_array** – shifted array

Return type **ndarray**

specialrange(val_min, val_max, steps, typ='exp')

Calculation of special point ranges.

Parameters

- **val_min** (float) – Starting value.
- **val_max** (float) – Ending value
• **steps** (int) – Number of steps.
• **typ** (str or float, optional) – Setting the kind of range-distribution. One can choose between
  - "exp": for exponential behavior
  - "log": for logarithmic behavior
  - "geo": for geometric behavior
  - "lin": for linear behavior
  - "quad": for quadratic behavior
  - "cub": for cubic behavior
  - float: here you can specify any exponent ("quad" would be equivalent to 2)

Default: "exp"

**Returns** Array containing the special range

**Return type** numpy.ndarray

### Examples

```py
>>> specialrange(1, 10, 4)
array([ 1. , 2.53034834, 5.23167968, 10. ])
```

---

**split_file_path** *(path, abs_path=False)*

Decompose a path to a file.

Decompose into the dir-path, the basename and the file-extension.

**Parameters**

- **path** (string) – string containing the path to a file
- **abs_path** (bool, optional) – convert the path to an absolute path. Default: False

**Returns** result – tuple containing the dir-path, basename and file-extension

**Return type** tuple of strings

---

**transform_points** *(points, xyz_func, **kwargs)*

Transform points with a given function “xyz_func”.

kwargs will be forwarded to “xyz_func”.

**Parameters**

- **points** (ndarray) – Array with all points positions.
- **xyz_func** (function) – the function transforming the points x_new, y_new, z_new = f(x_old, y_old, z_old, **kwargs)

**Returns** new_array – transformed array

**Return type** ndarray

---

**uncomment** *(line)*

Remove OGS comments from a given line of an OGS file.

Comments are indicated by “;”. The line is then splitted by whitespaces.

**Parameters** **line** (str) – given line
unique_rows (data, decimals=4, fast=True)

Unique made row-data with respect to given precision.

This is constructed to work best if point-pairs appear. The output is sorted like the input data. data needs to be 2D

Parameters

- **data** (*ndarray*) – 2D array containing the list of vectors that should be made unique
- **decimals** (*int, optional*) – Number of decimal places to round the ‘data’ to (default: 3). If decimals is negative, it specifies the number of positions to the left of the decimal point. This will not round the output, it is just for comparison of the vectors.
- **fast** (*bool, optional*) – If fast is True, the vector comparison is executed by a decimal comparison. If fast is False, all pairwise distances are calculated. Default: True

Returns

- **result** (*ndarray*) – 2D array of unique rows of data
- **ix** (*ndarray*) – index positions of output in input data (data[ix] = result) len(ix) = result.shape[0]
- **ixr** (*ndarray*) – reversed index positions of input in output data (result[ixr] = data) len(ixr) = data.shape[0]

Notes

This routine will preserve the order within the given array as effectively as possible. If you use it with a stack of 2 arrays and the first one is already unique, the resulting array will still have the first array at the beginning.

unique_rows_old (data, decimals=4)

Returns unique made data with respect to given precision in “decimals”.

The output is sorted like the input data. data needs to be 2D

Parameters

- **data** (*ndarray*) – 2D array containing the list of vectors that should be made unique

- **decimals** (*int, optional*) – Number of decimal places to round the ‘data’ to (default: 3). If decimals is negative, it specifies the number of positions to the left of the decimal point. This will not round the output, it is just for comparison of the vectors.

Returns

- **result** (*ndarray*) – 2D array of unique rows of data
- **ix** (*ndarray*) – index positions of output in input data (data[ix] = result) len(ix) = result.shape[0]
- **ixr** (*ndarray*) – reversed index positions of input in output data (result[ixr] = data) len(ixr) = data.shape[0]

Notes
This routine will preserve the order within the given array as effectively as possible. If you use it with a stack of 2 arrays and the first one is already unique, the resulting array will still have the first array at the beginning.

volume (typ, *pnt)
Volume of a OGS5 Meshelement.

Parameters

* typ (string) – OGS5 Meshelement type. Should be one of the following:
  - "line" : 1D element with 2 nodes
  - "tri" : 2D element with 3 nodes
  - "quad" : 2D element with 4 nodes
  - "tet" : 3D element with 4 nodes
  - "pyra" : 3D element with 5 nodes
  - "pris" : 3D element with 6 nodes
  - "hex" : 3D element with 8 nodes

* pnt (Node Coordinates pnt = (x_0, x_1, ...)) – List of points defining the Meshelement. A point is given as an (x,y,z) tuple and for each point, there can be a stack of points, if the volume should be calculated for multiple elements of the same type.

Returns Volume – Array containing the volumes of the give elements.

Return type ndarray
ogs5py Documentation, Release 1.1.1

ogs5py.tools.script

Script generator for ogs5py.

**Generator**

```python
_gen_script(ogs_class[, script_dir, ...])
```

Generate a python script for the given model.

**Helpers**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>formater(val)</td>
<td>Format values as string.</td>
</tr>
<tr>
<td>get_line(cont_line)</td>
<td>Create content line for the script.</td>
</tr>
<tr>
<td>tab(num)</td>
<td>Get tab indentation.</td>
</tr>
<tr>
<td>add_block_file(block_file, script[, ...])</td>
<td>Add block-file creation to script.</td>
</tr>
<tr>
<td>add_load_file(load_file, script[, ...])</td>
<td>Add a file to be loaded from a script.</td>
</tr>
<tr>
<td>add_list_file(list_file, script, typ[, ...])</td>
<td>Add a listed file to be loaded from a script.</td>
</tr>
</tbody>
</table>

```python
add_block_file(block_file, script[, ...])
```

Add block-file creation to script.

**Parameters**

- `block_file` (**BlockFile**) – BlockFile class to be added to the script
- `script` (**stream**) – given opened file for the script
- `ogs_cls_name` (**str**) – name of the model within the script

```python
add_list_file(list_file, script, typ[, ...])
```

Add a listed file to be loaded from a script.

**Parameters**

- `list_file` (**File**) – listed file that should be saved and then loaded from the script
- `script` (**stream**) – given opened file for the script
- `typ` (**str**) – typ of the list file
- `ogs_cls_name` (**str**) – name of the model within the script

```python
add_load_file(load_file, script[, ...])
```

Add a file to be loaded from a script.

**Parameters**

- `load_file` (**OGSFile**) – file that should be saved and then loaded from the script
- `script` (**stream**) – given opened file for the script
- `ogs_cls_name` (**str**) – name of the model within the script

```python
formater(val)
```

Format values as string.

```python
gen_script(ogs_class[, script_dir, ...])
```

Generate a python script for the given model.

Parameters

- `script_dir` (**str**) – output directory
- `script_name` (**str**) – name of the script
- `output_dir` (**str**) – output directory
- `separate_files` (**bool**) – separate files

Chapter 3. ogs5py API
Parameters

- **ogs_class** (*OGS*) – model class to be converted to a script
- **script_dir** (*str*) – target directory for the script
- **script_name** (*str*) – name for the script file (including .py ending)
- **ogs_cls_name** (*str*) – name of the model in the script
- **task_root** (*str*) – used task_root in the script
- **task_id** (*str*) – used task_id in the script
- **output_dir** (*str*) – used output_dir in the script
- **separate_files** (*list of str or None*) – list of files, that should be written to separate files and then loaded from the script

Notes

This will only create BlockFiles from the script. GLI and MSH files as well as every listed or line-wise file will be stored separately.

get_line(*cont_line*)
Create content line for the script.

  Parameters *cont_line* (*list of values*) – content line from a BlockFile

tab(*num*)
Get tab indentation.

  Parameters *num* (*int*) – indentation depth
ogs5py Documentation, Release 1.1.1

ogs5py.tools.types

type definitions for ogs5

GLI related Constants

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTY_GLI</td>
<td>empty gli dict</td>
</tr>
<tr>
<td>GLI_KEYS</td>
<td>ogs gli dict keys</td>
</tr>
<tr>
<td>GLI_KEY_LIST</td>
<td>gli main keys</td>
</tr>
<tr>
<td>EMPTY_PLY</td>
<td>empty ogs gli polyline dict</td>
</tr>
<tr>
<td>PLY_KEYS</td>
<td>ogs gli polyline keys</td>
</tr>
<tr>
<td>PLY_KEY_LIST</td>
<td>gli polyline key types</td>
</tr>
<tr>
<td>PLY_TYPES</td>
<td>gli polyline key types</td>
</tr>
<tr>
<td>EMPTY_SRF</td>
<td>empty ogs gli surface dict</td>
</tr>
<tr>
<td>SRF_KEYS</td>
<td>ogs gli surface keys</td>
</tr>
<tr>
<td>SRF_KEY_LIST</td>
<td>gli surface keys</td>
</tr>
<tr>
<td>SRF_TYPES</td>
<td>gli surface key types</td>
</tr>
<tr>
<td>EMPTY_VOL</td>
<td>empty ogs gli volume dict</td>
</tr>
<tr>
<td>VOL_KEYS</td>
<td>ogs gli volume keys</td>
</tr>
<tr>
<td>VOL_KEY_LIST</td>
<td>gli volume keys</td>
</tr>
<tr>
<td>VOL_TYPES</td>
<td>gli volume key types</td>
</tr>
</tbody>
</table>

MSH related Constants

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPTY_MSH</td>
<td>empty mesh dict</td>
</tr>
<tr>
<td>MESH_KEYS</td>
<td>ogs mesh dict-keys</td>
</tr>
<tr>
<td>MESH_DATA_KEYS</td>
<td>ogs mesh data keys</td>
</tr>
<tr>
<td>ELEM_1D</td>
<td>ogs element names</td>
</tr>
<tr>
<td>ELEM_2D</td>
<td>ogs element names</td>
</tr>
<tr>
<td>ELEM_3D</td>
<td>ogs element names</td>
</tr>
<tr>
<td>ELEM_DIM</td>
<td>ogs element names</td>
</tr>
<tr>
<td>ELEM_NAMES</td>
<td>ogs element names</td>
</tr>
<tr>
<td>ELEM_TYP</td>
<td>type code per element name</td>
</tr>
<tr>
<td>ELEM_TYP1D</td>
<td>type code per element name</td>
</tr>
<tr>
<td>ELEM_TYP2D</td>
<td>type code per element name</td>
</tr>
<tr>
<td>ELEM_TYP3D</td>
<td>type code per element name</td>
</tr>
<tr>
<td>VTK_TYP</td>
<td>vtk type codes per element name</td>
</tr>
<tr>
<td>MESHIO_NAMES</td>
<td>corresponding element names in meshio</td>
</tr>
<tr>
<td>NODE_NO</td>
<td>Node numbers per element name</td>
</tr>
</tbody>
</table>

General Constants

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRTYPE</td>
<td>alias of builtins.str</td>
</tr>
<tr>
<td>PCS_TYP</td>
<td>PCS types</td>
</tr>
<tr>
<td>PCS_EXT</td>
<td>PCS file extensions with _</td>
</tr>
<tr>
<td>PRIM_VAR</td>
<td>primary variables</td>
</tr>
<tr>
<td>PRIM_VAR_BY_PCS</td>
<td>primary variables per PCS</td>
</tr>
<tr>
<td>OGS_EXT</td>
<td>all ogs file extensions</td>
</tr>
<tr>
<td>MULTI_FILES</td>
<td>all ogs files that can occure multiple times</td>
</tr>
</tbody>
</table>

STRTYPE

alias of builtins.str
ELEMENT_KEYS = {'hex', 'line', 'pris', 'pyra', 'quad', 'tet', 'tri'}
  ogs element names

  Type  set

ELEM_1D = ['line']
  ogs element names

  Type  set

ELEM_2D = ['tri', 'quad']
  ogs element names

  Type  set

ELEM_3D = ['tet', 'pyra', 'pris', 'hex']
  ogs element names

  Type  set

ELEM_DIM = [['line'], ['tri', 'quad'], ['tet', 'pyra', 'pris', 'hex']]  
  ogs element names

  Type  set

ELEM_NAMES = ['line', 'tri', 'quad', 'tet', 'pyra', 'pris', 'hex']
  ogs element names

  Type  list

  type code per element name

  Type  dict

ELEM_TYP1D = {0: 'line', 'line': 0}
  type code per element name

  Type  dict

ELEM_TYP2D = {1: 'tri', 2: 'quad', 'tri': 1, 'quad': 2}
  type code per element name

  Type  dict

  type code per element name

  Type  dict

EMPTY_GLI = {'point_md': None, 'point_names': None, 'points': None, 'polylines': []}
  empty gli dict

  Type  dict

EMPTY_MSH = {'element_id': {}, 'elements': {}, 'material_id': {}, 'mesh_data': {}},
  empty mesh dict

  Type  dict

EMPTY_PLY = {'EPSILON': None, 'ID': None, 'MAT_GROUP': None, 'NAME': None, 'POINTS': None}
  empty ogs gli polyline dict

  Type  dict

EMPTY_SRF = {'EPSILON': None, 'ID': None, 'MAT_GROUP': None, 'NAME': None, 'POLYLINES': None}
  empty ogs gli surface dict

  Type  dict

EMPTY_VOL = {'LAYER': None, 'MAT_GROUP': None, 'NAME': None, 'SURFACES': None, 'TYPE': None}
  empty ogs gli volume dict

  Type  dict
Type  dict

GLI_KEYS = {'point_md', 'point_names', 'points', 'polylines', 'surfaces', 'volumes'}
ogs gli dict keys

Type  set

GLI_KEY_LIST = ['#POINTS', '#POLYLINE', '#SURFACE', '#VOLUME', '#STOP']
gli main keys

Type  list

MESHIO_NAMES = ['line', 'triangle', 'quad', 'tetra', 'pyramid', 'wedge', 'hexahedron']
corresponding element names in meshio

Type  list

MESH_DATA_KEYS = {'AXISYMMETRY', 'CROSS_SECTION', 'GEO_NAME', 'GEO_TYPE', 'LAYER', 'PCS_TYPE'}
ogs mesh data keys

Type  set

MESH_KEYS = {'element_id', 'elements', 'material_id', 'mesh_data', 'nodes'}
ogs mesh dict-keys

Type  set

MULTI_FILES = ['mpd', 'gli_ext', 'rfr', 'gem_init', 'asc']
all ogs files that can occur multiple times

Type  list

NODE_NO = {0: 2, 1: 3, 2: 4, 3: 4, 4: 5, 5: 6, 6: 8, 'line': 2, 'tri': 3, 'quad': 4, 'tet': 4, 'pyra': 5, 'pris': 6, 'hex': 8}
Node numbers per element name

Type  dict

OGS_EXT = ['.msh', '.gli', '.ddc', '.pcs', '.rfd', '.cct', '.fct', '.bc', '.ic', '.st', 
all ogs file extensions

Type  list

PCS_EXT = ['', '_GROUNDWATER_FLOW', '_LIQUID_FLOW', '_RICHARDS_FLOW', '_AIR_FLOW', '_MULTI_PHASE_FLOW', '_PS_GLOBAL', 
PCS file extensions with _

Type  list

PCS_TYP = ['', 'GROUNDWATER_FLOW', 'LIQUID_FLOW', 'RICHARDS_FLOW', 'AIR_FLOW', 'MULTI_PHASE_FLOW', 'FLUID_MOMENTUM', 'RANDOM_WALK', 'NO_PCS', 'TNEQ', 'TES', 'DEFORMATION_SINGLEFLOW_MONO', 'MULTI_COMPONENTIAL_FLOW'
PCS types

Type  list

PLY_KEYS = {'EPSILON', 'ID', 'MAT_GROUP', 'NAME', 'POINTS', 'POINT_VECTOR', 'TYPE'}
ogs gli polyline keys

Type  set

PLY_KEY_LIST = ['ID', 'NAME', 'TYPE', 'EPSILON', 'MAT_GROUP', 'POINTS', 'POINT_VECTOR']
gli polyline keys

Type  list

PLY_TYPES = [<class 'int'>, <class 'str'>, <class 'int'>, <class 'float'>, <class 'int'>]
gli polyline key types

Type  list

PRIM_VAR = ['', 'HEAD', ['PRESSURE1'], ['PRESSURE1'], ['PRESSURE1', 'TEMPERATURE1']
primary variables

Type  list
PRIM_VAR_BY_PCS = {'': [''], 'AIR_FLOW': ['PRESSURE1', 'TEMPERATURE1'], 'DEFORMATION': []}
        primary variables per PCS
        Type dict

SRF_KEYS = {'EPSILON', 'ID', 'MAT_GROUP', 'NAME', 'POLYLINES', 'TIN', 'TYPE'}
        ogs gli surface keys
        Type set

SRF_KEY_LIST = ['ID', 'NAME', 'EPSILON', 'TYPE', 'TIN', 'MAT_GROUP', 'POLYLINES']
        gli surface keys
        Type list

SRF_TYPES = [class 'int'], class 'str'], class 'float'], class 'int'], class 'str']
        gli surface key types
        Type list

VOL_KEYS = {'LAYER', 'MAT_GROUP', 'NAME', 'SURFACES', 'TYPE'}
        ogs gli volume keys
        Type set

VOL_KEY_LIST = ['NAME', 'TYPE', 'SURFACES', 'MAT_GROUP', 'LAYER']
        gli volume keys
        Type list

VOL_TYPES = [class 'str'], class 'str'], class 'list'], class 'str'], class 'int']
        gli volume key types
        Type list

        vtk type codes per element name
        Type dict
ogs5py Documentation, Release 1.1.1

ogs5py.tools.download

Downloader for ogs5.

Downloader

A downloading routine to get the OSG5 executable.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>download_ogs</td>
<td>Download the OGS5 executable.</td>
</tr>
<tr>
<td>add_exe</td>
<td>Add an OGS5 exe to OGS5PY_CONFIG.</td>
</tr>
<tr>
<td>reset_download</td>
<td>Reset all downloads in OGS5PY_CONFIG.</td>
</tr>
<tr>
<td>OGS5PY_CONFIG</td>
<td>Standard config path for ogs5py.</td>
</tr>
</tbody>
</table>

**add_exe** *(ogs_exe[, dest_name]*)

Add an OGS5 exe to OGS5PY_CONFIG.

Parameters

- **ogs_exe** *(str)* – Path to the ogs executable to be copied.
- **dest_name** *(str, optional)* – Destination file name. Default: basename of ogs_exe

Returns **dest** – If an OGS5 executable was successfully copied, the file-path is returned.

Return type **str**

**download_ogs** *(version='5.7', system=None, path='/home/docs/.config/ogs5py', name=None, build=None)*

Download the OGS5 executable.

Parameters

- **version** *(str, optional)* – Version to download (“5.7”, “5.8”, “latest” or “stable”). Default: “5.7”
- **system** *(str, optional)* – Target system (Linux, Windows, Darwin). Default: platform.system()
- **path** *(str, optional)* – Destination path. Default: OGS5PY_CONFIG
- **name** *(str, optional)* – Destination file name. Default “ogs[.exe]”
- **build** *(str, optional)* – If system is “Linux” and version is “latest” or “stable”, you can select a certain build from the ogs 5 builds:
  - “BRNS”: Biogeochemical Reaction Network Simulator
  - “FEM”: Finite Element Method
  - “GEMS”: Gibbs Energy Minimization Solver
  - “IPQC”: IPhreeqc
  - “LIS”: Library of Iterative Solvers
  - “MKL”: Intel Math Kernel Library
  - “MPI”: Message Passing Interface
  - “PETSC”: Portable, Extensible Toolkit for Scientific Computation
  - “PETSC_GEMS”: PETSC and GEMS
  - “PQC”: PHREEQC
  - “SP”: Sparse solver
Returns **dest** – If an OGS executable was successfully downloaded, the file-path is returned.

**Return type** `str`

**Notes**

There is only an executable on “Darwin” for version “5.7”.

Taken from:


---

**get_links** `(url, ext, build=None)`

Get links from url ending with ext and containing build.

**reset_download** ()

Reset all downloads in **OGS5PY_CONFIG**.

**OGS5PY_CONFIG** = `'./home/docs/.config/ogs5py'`

Standard config path for ogs5py.

**Type** `str`
ogs5py Documentation, Release 1.1.1

ogs5py.tools.output

Tools for ogs5py output files (independent from VTK package).

get_output_files(task_root, task_id, pcs=None, typ='VTK', element=None)

Get a list of output file paths.

Parameters

- **task_root** (string) – string containing the path to the directory containing the ogs output
- **task_id** (string) – string containing the file name of the ogs task without extension
- **pcs** (string or None, optional) – specify the PCS type that should be collected Possible values are:
  - None/"" (no PCS_TYPE specified in *out)
  - "NO_PCS"
  - "GROUNDWATER_FLOW"
  - "LIQUID_FLOW"
  - "RICHARDS_FLOW"
  - "AIR_FLOW"
  - "MULTI_PHASE_FLOW"
  - "PS_GLOBAL"
  - "HEAT_TRANSPORT"
  - "DEFORMATION"
  - "MASS_TRANSPORT"
  - "OVERLAND_FLOW"
  - "FLUID_MOMENTUM"
  - "RANDOM_WALK"
  Default : None
- **typ** (string, optional) – Type of the output ("VTK", “PVD”, “TEC_POINT” or “TEC_POLYLINE”). Default : “VTK”
- **element** (string or None, optional) – For tecplot output you can specify the name of the output element. (Point-name of Line-name from GLI file) Default: None

readpvd_single(infile)

Read a paraview pvd file.

Convert all concerned files to a dictionary containing their data.

split_ply_path(infile, task_id=None, line_name=None, PCS_name=None, split_extra=False)

Retrive ogs-infos from filename for tecplot-polyline output.

{id}_ply_{line}_{t{n}[}_{PCS+extra]}.tec

split_pnt_path(infile, task_id=None, pnt_name=None, PCS_name=None, split_extra=False, guess_PCS=False)

Retrive ogs-infos from filename for tecplot-polyline output.

{id}_time_{pnt}[_{PCS+extra}].tec
ogs5py.tools.vtk_viewer

Viewer for a vtk file.

**show_vtk** (*vtkfile, log_scale=False*)

Display a given mesh colored by its material ID.

**Parameters**

- **vtkfile** (*str*) – Path to the vtk/vtu file to show.
- **log_scale** (*bool, optional*) – State if the data should be shown in log scale.
  Default: False

**Notes**

This routine needs “mayavi” to display the mesh. (see here: https://github.com/enthought/mayavi)
## PYTHON MODULE INDEX

<table>
<thead>
<tr>
<th>Module Path</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ogs5py</td>
<td>13</td>
</tr>
<tr>
<td>ogs5py.fileclasses</td>
<td>25</td>
</tr>
<tr>
<td>ogs5py.fileclasses.asc</td>
<td>36</td>
</tr>
<tr>
<td>ogs5py.fileclasses.base</td>
<td>26</td>
</tr>
<tr>
<td>ogs5py.fileclasses.bc</td>
<td>39</td>
</tr>
<tr>
<td>ogs5py.fileclasses.cct</td>
<td>46</td>
</tr>
<tr>
<td>ogs5py.fileclasses.ddc</td>
<td>52</td>
</tr>
<tr>
<td>ogs5py.fileclasses.fct</td>
<td>58</td>
</tr>
<tr>
<td>ogs5py.fileclasses.gem</td>
<td>64</td>
</tr>
<tr>
<td>ogs5py.fileclasses.gli</td>
<td>72</td>
</tr>
<tr>
<td>ogs5py.fileclasses.gli.generator</td>
<td>82</td>
</tr>
<tr>
<td>ogs5py.fileclasses.ic</td>
<td>84</td>
</tr>
<tr>
<td>ogs5py.fileclasses.krc</td>
<td>93</td>
</tr>
<tr>
<td>ogs5py.fileclasses.mcp</td>
<td>101</td>
</tr>
<tr>
<td>ogs5py.fileclasses.mfp</td>
<td>108</td>
</tr>
<tr>
<td>ogs5py.fileclasses.mmp</td>
<td>115</td>
</tr>
<tr>
<td>ogs5py.fileclasses.mpd</td>
<td>123</td>
</tr>
<tr>
<td>ogs5py.fileclasses.msh</td>
<td>130</td>
</tr>
<tr>
<td>ogs5py.fileclasses.msh.generator</td>
<td>141</td>
</tr>
<tr>
<td>ogs5py.fileclasses.msp</td>
<td>146</td>
</tr>
<tr>
<td>ogs5py.fileclasses.num</td>
<td>153</td>
</tr>
<tr>
<td>ogs5py.fileclasses.out</td>
<td>160</td>
</tr>
<tr>
<td>ogs5py.fileclasses.pcs</td>
<td>167</td>
</tr>
<tr>
<td>ogs5py.fileclasses.pct</td>
<td>174</td>
</tr>
<tr>
<td>ogs5py.fileclasses.pqc</td>
<td>176</td>
</tr>
<tr>
<td>ogs5py.fileclasses.rei</td>
<td>180</td>
</tr>
<tr>
<td>ogs5py.fileclasses.rfd</td>
<td>187</td>
</tr>
<tr>
<td>ogs5py.fileclasses.st</td>
<td>193</td>
</tr>
<tr>
<td>ogs5py.fileclasses.tim</td>
<td>200</td>
</tr>
<tr>
<td>ogs5py.ogs</td>
<td>16</td>
</tr>
<tr>
<td>ogs5py.reader</td>
<td>207</td>
</tr>
<tr>
<td>ogs5py.tools</td>
<td>211</td>
</tr>
<tr>
<td>ogs5py.tools.download</td>
<td>226</td>
</tr>
<tr>
<td>ogs5py.tools.output</td>
<td>228</td>
</tr>
<tr>
<td>ogs5py.tools.script</td>
<td>220</td>
</tr>
<tr>
<td>ogs5py.tools.tools</td>
<td>212</td>
</tr>
<tr>
<td>ogs5py.tools.types</td>
<td>222</td>
</tr>
<tr>
<td>ogs5py.tools.vtk_viewer</td>
<td>229</td>
</tr>
</tbody>
</table>
Symbols

__call__() (GLI method), 74
__call__() (MSH method), 132

A
add() (MultiFile method), 35
add_asc() (OGS method), 18
add_block() (BC method), 40
add_block() (BlockFile method), 27
add_block() (CCT method), 47
add_block() (DDC method), 53
add_block() (FCT method), 59
add_block() (GEM method), 65
add_block() (IC method), 85
add_block() (KRC method), 96
add_block() (MCP method), 103
add_block() (MFP method), 110
add_block() (MMP method), 117
add_block() (MPD method), 124
add_block() (MSP method), 147
add_block() (NUM method), 154
add_block() (OUT method), 162
add_block() (PCS method), 169
add_block() (REI method), 181
add_block() (RFD method), 188
add_block() (ST method), 194
add_block() (TIM method), 201
add_block_file() (in module ogs5py.tools.script), 220
add_content() (BC method), 41
add_content() (BlockFile method), 27
add_content() (CCT method), 47
add_content() (DDC method), 53
add_content() (FCT method), 59
add_content() (GEM method), 66
add_content() (IC method), 86
add_content() (KRC method), 96
add_content() (MCP method), 103
add_content() (MFP method), 110
add_content() (MMP method), 118
add_content() (MPD method), 125
add_content() (MSP method), 148
add_content() (NUM method), 155
add_content() (OUT method), 162
add_content() (PCS method), 169
add_content() (REI method), 182
add_content() (RFD method), 188
add_content() (ST method), 195
add_content() (TIM method), 202
add_copy_file() (OGS method), 18
add_copy_link() (ASC method), 37
add_copy_link() (BC method), 41
add_copy_link() (BlockFile method), 27
add_copy_link() (CCT method), 48
add_copy_link() (DDC method), 54
add_copy_link() (FCT method), 60
add_copy_link() (File method), 32
add_copy_link() (GEM method), 66
add_copy_link() (GLI method), 74
add_copy_link() (GLIext method), 80
add_copy_link() (IC method), 86
add_copy_link() (KRC method), 97
add_copy_link() (LineFile method), 34
add_copy_link() (MCP method), 103
add_copy_link() (MFP method), 110
add_copy_link() (MMP method), 118
add_copy_link() (MPD method), 125
add_copy_link() (MSH method), 132
add_copy_link() (MSP method), 148
add_copy_link() (NUM method), 155
add_copy_link() (OUT method), 162
add_copy_link() (PCS method), 170
add_copy_link() (PCT method), 174
add_copy_link() (PQC method), 176
add_copy_link() (PQCdat method), 178
add_copy_link() (REI method), 182
add_copy_link() (RFD method), 189
add_copy_link() (RFR method), 91
add_copy_link() (ST method), 195
add_copy_link() (TIM method), 202
add_exe() (in module ogs5py.tools.download), 226
add_gem_init() (OGS method), 18
add_gli_ext() (OGS method), 18
add_list_file() (in module ogs5py.tools.script), 220
add_load_file() (in module ogs5py.tools.script), 220
add_main_keyword() (BC method), 41
add_main_keyword() (BlockFile method), 28
add_main_keyword() (CCT method), 48
add_main_keyword() (DDC method), 54
add_main_keyword() (FCT method), 60
add_main_keyword() (GEM method), 66
add_main_keyword() (IC method), 86
add_main_keyword() (KRC method), 97
add_main_keyword() (MCP method), 103
add_main_keyword() (MFP method), 110
add_main_keyword() (MMP method), 118
add_main_keyword() (MPD method), 125
add_main_keyword() (MSP method), 148
add_main_keyword() (NUM method), 155
add_main_keyword() (OUT method), 163
add_main_keyword() (PCS method), 170
add_main_keyword() (REI method), 182
add_main_keyword() (RFD method), 189
add_main_keyword() (ST method), 195
add_main_keyword() (TIM method), 202
add_mpd() (OGS method), 18
add_multi_content() (BC method), 41
add_multi_content() (BlockFile method), 28
add_multi_content() (CCT method), 48
add_multi_content() (DDC method), 54
add_multi_content() (FCT method), 60
add_multi_content() (GEM method), 66
add_multi_content() (IC method), 86
add_multi_content() (KRC method), 97
add_multi_content() (MCP method), 104
add_multi_content() (MFP method), 111
add_multi_content() (MMP method), 118
add_multi_content() (MPD method), 125
add_multi_content() (MSP method), 148
add_multi_content() (NUM method), 156
add_multi_content() (OUT method), 163
add_multi_content() (PCS method), 170
add_multi_content() (REI method), 182
add_multi_content() (RFD method), 189
add_multi_content() (ST method), 196
add_multi_content() (TIM method), 202
add_points() (GLI method), 74
add_polyline() (GLI method), 75
add_rfr() (OGS method), 18
add_sub_keyword() (BC method), 42
add_sub_keyword() (BlockFile method), 28
add_sub_keyword() (CCT attribute), 51
add_sub_keyword() (DDC attribute), 57
add_sub_keyword() (FCT attribute), 63
add_sub_keyword() (KRC attribute), 100
add_sub_keyword() (MCP attribute), 107
add_sub_keyword() (MFP attribute), 114
add_sub_keyword() (MMP attribute), 121
add_sub_keyword() (MPD attribute), 128
add_sub_keyword() (MSP attribute), 138
add_sub_keyword() (NUM attribute), 159
add_sub_keyword() (OUT attribute), 166
add_sub_keyword() (PCS attribute), 173
add_sub_keyword() (REI attribute), 186
add_sub_keyword() (RFD attribute), 192
add_sub_keyword() (ST attribute), 199
add_sub_keyword() (TIM attribute), 205
BlockFile (class in ogs5py.fileclasses.base), 26

B
BC (class in ogs5py.fileclasses.bc), 39
block (MSH attribute), 138
block_adapter3D() (in module ogs5py.fileclasses.msh.generator), 141
block_no (BC attribute), 44
block_no (BlockFile attribute), 31
block_no (CCT attribute), 51
block_no (DDC attribute), 57
block_no (FCT attribute), 63
block_no (GEM attribute), 69
block_no (IC attribute), 89
block_no (KRC attribute), 100
block_no (MCP attribute), 107
block_no (MFP attribute), 114
block_no (MMP attribute), 121
block_no (MPD attribute), 128
block_no (MSP attribute), 138
block_no (NUM attribute), 159
block_no (OUT attribute), 166
block_no (PCS attribute), 173
block_no (REI attribute), 186
block_no (RFD attribute), 192
block_no (ST attribute), 199
block_no (TIM attribute), 205
BlockFile (class in ogs5py.fileclasses.base), 26

Index
bot_com (OGS attribute), 24
by_id() (in module ogs5py.tools.tools), 213
C
cct (class in ogs5py.fileclasses.cct), 46
center (MSH attribute), 138
centroid() (in module ogs5py.tools.tools), 213
centroids (MSH attribute), 138
centroids_flat (MSH attribute), 139
check() (ASC method), 37
check() (BC method), 42
check() (BlockFile method), 29
check() (CCT method), 49
check() (DDC method), 55
check() (FCT method), 61
check() (File method), 32
check() (GEM method), 67
check() (GEMInit method), 71
check() (GLI method), 75
check() (GLIext method), 81
check() (IC method), 87
check() (KRC method), 98
check() (LineFile method), 34
check() (MCP method), 105
check() (MFP method), 112
check() (MMP method), 119
check() (MPD method), 126
check() (MSH method), 132
check() (MSP method), 149
check() (NUM method), 157
check() (OUT method), 164
check() (PCS method), 171
check() (PCT method), 174
click() (PQC method), 177
click() (PQCdat method), 178
click() (REI method), 183
click() (RFD method), 190
click() (RFR method), 91
click() (ST method), 197
click() (TIM method), 203
close() (Output method), 213
combine_mesh() (MSH method), 132
CROSS_SECTION (MSH attribute), 136
D
data (RFR attribute), 92
ddc (class in ogs5py.fileclasses.ddc), 52
del_asc() (OGS method), 18
del_block() (BC method), 43
del_block() (BlockFile method), 29
del_block() (CCT method), 49
del_block() (DDC method), 55
del_block() (FCT method), 61
del_block() (GEM method), 68
del_block() (IC method), 87
del_block() (KRC method), 98
del_block() (MCP method), 105
del_block() (MFP method), 112
del_block() (MMP method), 119
del_block() (MPD method), 126
del_block() (MSP method), 150
del_block() (NUM method), 157
del_block() (OUT method), 164
del_block() (PCS method), 171
del_block() (REI method), 184
del_block() (RFD method), 190
del_block() (ST method), 197
del_block() (TIM method), 204
del_content() (BC method), 43
del_content() (BlockFile method), 29
del_content() (CCT method), 49
del_content() (DDC method), 55
del_content() (FCT method), 61
del_content() (GEM method), 68
del_content() (IC method), 88
del_content() (KRC method), 98
del_content() (MCP method), 105
del_content() (MFP method), 112
del_content() (MMP method), 120
del_content() (MPD method), 127
del_content() (MSP method), 150
del_content() (NUM method), 157
del_content() (OUT method), 164
del_content() (PCS method), 171
del_content() (REI method), 184
del_content() (RFD method), 190
del_content() (ST method), 197
del_content() (TIM method), 204
del_copy_file() (OGS method), 18
del_copy_link() (ASC method), 37
del_copy_link() (BC method), 43
del_copy_link() (BlockFile method), 29
del_copy_link() (CCT method), 50
del_copy_link() (DDC method), 56
del_copy_link() (FCT method), 62
del_copy_link() (File method), 32
del_copy_link() (GEM method), 68
del_copy_link() (GLI method), 76
del_copy_link() (GLIext method), 81
del_copy_link() (IC method), 88
del_copy_link() (KRC method), 99
del_copy_link() (LineFile method), 34
del_copy_link() (MCP method), 105
del_copy_link() (MFP method), 112
del_copy_link() (MMP method), 120
del_copy_link() (MPD method), 127
del_copy_link() (MSP method), 132
del_copy_link() (MSH method), 132
del_copy_link() (NUM method), 157
del_copy_link() (OUT method), 164
del_copy_link() (PCS method), 171
del_copy_link() (PCT method), 175
del_copy_link() (PQC method), 177
del_copy_link() (PQCdat method), 179
del_copy_link() (REI method), 184
del_copy_link() (RFD method), 191

delblock() (BlockFile method), 29
delblock() (CCT method), 49
delblock() (DDC method), 55
delblock() (FCT method), 61
delblock() (GEM method), 68
delblock() (IC method), 87
delblock() (KRC method), 98
delblock() (MCP method), 105
delblock() (MFP method), 112
del_copy_link() *(RFR method)*, 91
del_copy_link() *(ST method)*, 197
del_copy_link() *(TIM method)*, 204
del_gem_init() *(OGS method)*, 18
del_gli_ext() *(OGS method)*, 19
del_main_keyword() *(BC method)*, 43
del_main_keyword() *(BlockFile method)*, 29
del_main_keyword() *(CCCT method)*, 50
del_main_keyword() *(DDC method)*, 56
del_main_keyword() *(FCT method)*, 62
del_main_keyword() *(GEM method)*, 68
del_main_keyword() *(IC method)*, 88
del_main_keyword() *(KRC method)*, 99
del_main_keyword() *(MCP method)*, 105
del_main_keyword() *(MFP method)*, 112
del_main_keyword() *(MPD method)*, 127
del_main_keyword() *(MSP method)*, 150
del_main_keyword() *(NUM method)*, 157
del_main_keyword() *(OUT method)*, 164
del_main_keyword() *(PCS method)*, 172
del_main_keyword() *(REI method)*, 184
del_main_keyword() *(RFD method)*, 191
del_main_keyword() *(ST method)*, 197
del_main_keyword() *(TIM method)*, 204
del_mpd() *(OGS method)*, 19
del_rfr() *(OGS method)*, 19
del_sub_keyword() *(BC method)*, 43
del_sub_keyword() *(BlockFile method)*, 29
del_sub_keyword() *(CCCT method)*, 50
del_sub_keyword() *(DDC method)*, 56
del_sub_keyword() *(FCT method)*, 62
del_sub_keyword() *(GEM method)*, 68
del_sub_keyword() *(IC method)*, 88
del_sub_keyword() *(KRC method)*, 99
del_sub_keyword() *(MCP method)*, 105
del_sub_keyword() *(MFP method)*, 112
del_sub_keyword() *(MPD method)*, 127
del_sub_keyword() *(MSP method)*, 150
del_sub_keyword() *(NUM method)*, 157
del_sub_keyword() *(OUT method)*, 164
del_sub_keyword() *(PCS method)*, 172
del_sub_keyword() *(REI method)*, 184
del_sub_keyword() *(RFD method)*, 191
del_sub_keyword() *(ST method)*, 197
del_sub_keyword() *(TIM method)*, 204
delete() *(MultiFile method)*, 35
download_ogs() *(in module ogs5py.tools.download)*, 226

E
ELEM_1D *(in module ogs5py.tools.types)*, 223
ELEM_2D *(in module ogs5py.tools.types)*, 223
ELEM_3D *(in module ogs5py.tools.types)*, 223
ELEM_DIM *(in module ogs5py.tools.types)*, 223
ELEM_NAMES *(in module ogs5py.tools.types)*, 223
ELEM_TYP *(in module ogs5py.tools.types)*, 223
ELEM_TYP1D *(in module ogs5py.tools.types)*, 223
ELEM_TYP2D *(in module ogs5py.tools.types)*, 223
ELEM_TYP3D *(in module ogs5py.tools.types)*, 223
ELEMENT_ID *(MSH attribute)*, 137
ELEMENT_KEYS *(in module ogs5py.tools.types)*, 222
ELEMENT_NO *(MSH attribute)*, 137
ELEMENT_TYPES *(MSH attribute)*, 137
ELEMENTS *(MSH attribute)*, 137
EMPTY_GLI *(in module ogs5py.tools.types)*, 223
EMPTY_MSH *(in module ogs5py.tools.types)*, 223
EMPTY_FLY *(in module ogs5py.tools.types)*, 223
EMPTY_SRF *(in module ogs5py.tools.types)*, 223
EMPTY_VOL *(in module ogs5py.tools.types)*, 223
export_mesh() *(MSH method)*, 132

F
FCT *(class in ogs5py.fileclasses.fct)*, 58
File *(class in ogs5py.fileclasses.base)*, 31
file_ext *(GEMinit attribute)*, 71
file_name *(ASC attribute)*, 37
file_name *(BC attribute)*, 44
file_name *(BlockFile attribute)*, 31
file_name *(CCT attribute)*, 51
file_name *(DDC attribute)*, 57
file_name *(FCT attribute)*, 63
file_name *(File attribute)*, 32
file_name *(GEM attribute)*, 69
file_name *(GLI attribute)*, 79
file_name *(GLIext attribute)*, 81
file_name *(IC attribute)*, 89
file_name *(KRC attribute)*, 100
file_name *(LineFile attribute)*, 34
file_name *(MCP attribute)*, 107
file_name *(MFP attribute)*, 114
file_name *(MMP attribute)*, 121
file_name *(MSP attribute)*, 128
file_name *(MSH attribute)*, 139
file_name *(MSP attribute)*, 152
file_name *(NUM attribute)*, 159
file_name *(OUT attribute)*, 166
file_name *(PCS attribute)*, 173
file_name *(PQC attribute)*, 175
file_name *(PQCdat attribute)*, 177
file_name *(REI attribute)*, 186
file_name *(RFD attribute)*, 192
file_name *(RFR attribute)*, 92
file_name *(ST attribute)*, 199
file_name *(TIM attribute)*, 205
file_names *(GEMinit attribute)*, 71
file_path *(ASC attribute)*, 37
file_path *(BC attribute)*, 45
file_path *(BlockFile attribute)*, 31
file_path *(CCT attribute)*, 51
file_path *(DDC attribute)*, 57
file_path *(FCT attribute)*, 63
file_path *(File attribute)*, 33
file_path *(GEM attribute)*, 69
format_content_line() (in module ogs5py.tools.tools), 214
format_dict() (in module ogs5py.tools.tools), 214
formater() (in module ogs5py.tools.script), 220

G
GEM (class in ogs5py.fileclasses.gem), 64
GEMinit (class in ogs5py.fileclasses.gem), 70
gen_script() (in module ogs5py.tools.script), 220
gen_script() (OGS method), 19
generate() (GLI method), 76
generate() (MCP attribute), 133
generate_time() (in module ogs5py.tools.tools), 214

GEO_NAME (MSH attribute), 137
GEO_TYPE (MSH attribute), 137
get_block() (BC method), 43
get_block() (BlockFile method), 30
get_block() (CCT method), 50
get_block() (DDC method), 56
get_block() (FCT method), 62
get_block() (GEM method), 68
get_block() (IC method), 88
get_block() (KRC method), 99
get_block() (MCP method), 106
get_block() (MFP method), 113
get_block() (MMP method), 120
get_block() (MPD method), 127
get_block() (MSP method), 150
get_block() (NUM method), 158
get_block() (OUT method), 165
get_block() (PCS method), 172
get_block() (REI method), 184
get_block() (RFD method), 191
get_block() (ST method), 198
get_block() (TIM method), 204
get_block_no() (BC method), 44
get_block_no() (BlockFile method), 30
get_block_no() (CCT method), 50
get_block_no() (DDC method), 56
get_block_no() (FCT method), 62
get_block_no() (GEM method), 68
get_block_no() (IC method), 88
get_block_no() (KRC method), 99
get_block_no() (MCP method), 106
get_block_no() (MFP method), 113
get_block_no() (MMP method), 120
get_block_no() (MPD method), 127
get_block_no() (MSP method), 151
get_block_no() (NUM method), 158
get_block_no() (OUT method), 165
get_block_no() (PCS method), 172
get_block_no() (REI method), 185
get_block_no() (RFD method), 191
get_block_no() (ST method), 198
get_block_no() (TIM method), 205
gen_file_type() (ASC method), 37
gen_file_type() (BC method), 44
get_file_type() (BlockFile method), 30
get_file_type() (CCT method), 50
get_file_type() (DDC method), 56
get_file_type() (FCT method), 62
get_file_type() (File method), 32
get_file_type() (GEM method), 68
get_file_type() (GEMinit method), 71
get_file_type() (GLI method), 76
get_file_type() (GLIext method), 81
get_file_type() (IC method), 88
get_file_type() (KRC method), 99
get_file_type() (LineFile method), 34
get_file_type() (MCP method), 106
get_file_type() (MFP method), 113
get_file_type() (MMP method), 120
get_file_type() (MPD method), 127
get_file_type() (MSH method), 133
get_file_type() (MSP method), 151
get_file_type() (NUM method), 158
get_file_type() (OUT method), 165
get_file_type() (PCS method), 172
get_file_type() (FCT method), 175
get_file_type() (PQC method), 177
get_file_type() (PQCdat method), 179
get_file_type() (REI method), 185
get_file_type() (RFD method), 191
get_file_type() (RFR method), 91
get_file_type() (ST method), 198
get_file_type() (TIM method), 205
get_key() (in module ogs5py.tools.tools), 214
get_links() (in module ogs5py.tools.script), 221
get_multi_keys() (BC method), 44
get_multi_keys() (BlockFile method), 30
get_multi_keys() (CCT method), 50
get_multi_keys() (DDC method), 56
get_multi_keys() (FCT method), 62
get_multi_keys() (GEM method), 69
get_multi_keys() (IC method), 88
get_multi_keys() (KRC method), 99
get_multi_keys() (MCP method), 106
get_multi_keys() (MFP method), 113
get_multi_keys() (MMP method), 127
get_multi_keys() (MSP method), 151
get_multi_keys() (NUM method), 158
get_multi_keys() (BC method), 45
get_multi_keys() (BlockFile attribute), 31
get_multi_keys() (CCT attribute), 51
get_multi_keys() (DDC attribute), 57
get_multi_keys() (FCT attribute), 63
get_multi_keys() (File attribute), 33
get_multi_keys() (GEM attribute), 70
get_multi_keys() (GEMinit attribute), 71
get_multi_keys() (GLI attribute), 79
get_multi_keys() (GLIext attribute), 81
get_multi_keys() (IC attribute), 90
get_multi_keys() (KRC attribute), 100
get_multi_keys() (LineFile attribute), 34
get_multi_keys() (MCP attribute), 107
get_multi_keys() (MFP attribute), 114
get_multi_keys() (MPD attribute), 122
get_multi_keys() (MSP attribute), 129
get_multi_keys() (MSH attribute), 139
get_multi_keys() (MSP attribute), 152
get_multi_keys() (NUM attribute), 159

GLIext (class in ogs5py.fileclasses.gli), 80
gmsh() (in module ogs5py.fileclasses.msh.generator), 141
grid_adapter2D() (in module ogs5py.fileclasses.msh.generator), 142
grid_adapter3D() (in module ogs5py.fileclasses.msh.generator), 143
guess_type() (in module ogs5py.tools.tools), 214

H
has_output_dir (OGS attribute), 24
hull_deform() (in module ogs5py.tools.tools), 214

I
IC (class in ogs5py.fileclasses.ic), 84

id (MultiFile attribute), 35
import_mesh() (MSH method), 134
is_block_unique() (BC method), 44
is_block_unique() (BlockFile method), 30
is_block_unique() (CCT method), 50
is_block_unique() (DDC method), 56
is_block_unique() (FCT method), 62
is_block_unique() (GEM method), 69
is_block_unique() (IC method), 89
is_block_unique() (KRC method), 99
is_block_unique() (MCP method), 106
is_block_unique() (MFP method), 113
is_block_unique() (MMP method), 121
is_block_unique() (MPD method), 128
is_block_unique() (MSP method), 151
is_block_unique() (NUM method), 158
is_block_unique() (BC attribute), 45
is_block_unique() (BlockFile attribute), 31
is_block_unique() (CCT attribute), 51
is_block_unique() (DDC attribute), 57
is_block_unique() (FCT attribute), 63
is_block_unique() (File attribute), 33
is_block_unique() (GEM attribute), 70
is_block_unique() (GEMinit attribute), 71
is_block_unique() (GLI attribute), 79
is_block_unique() (GLIext attribute), 81
is_block_unique() (IC attribute), 90
is_block_unique() (KRC attribute), 100
is_block_unique() (LineFile attribute), 34
is_block_unique() (MCP attribute), 107
is_block_unique() (MFP attribute), 114
is_block_unique() (MPD attribute), 122
is_block_unique() (MSP attribute), 129
is_block_unique() (MSH attribute), 139
is_block_unique() (MSP attribute), 152
is_block_unique() (NUM attribute), 159
is_empty (OUT attribute), 166
is_empty (PCS attribute), 173
is_empty (PCT attribute), 175
is_empty (PQC attribute), 177
is_empty (PQCdata attribute), 179
is_empty (REI attribute), 186
is_empty (RFD attribute), 192
is_empty (RF attribute), 92
is_empty (ST attribute), 199
is_empty (TIM attribute), 206
is_key (in module ogsSpy.fileclasses.tools), 215
is_mkey (in module ogsSpy.tools.tools), 215
is_skey (in module ogsSpy.tools.tools), 215
is_str_array (in module ogsSpy.tools.tools), 215

K
KRC (class in ogsSpy.fileclasses.krc), 93

L
LAYER (MSH attribute), 138
LineFile (class in ogsSpy.fileclasses.base), 33
load () (GLI method), 76
load () (MSH method), 134
load_model () (OGS method), 19

M
MATERIAL_ID (MSH attribute), 138
MATERIAL_ID_flat (MSH attribute), 138
MCP (class in ogsSpy.fileclasses.mcp), 101
MESH_DATA_KEYS (in module ogsSpy.tools.types), 224
MESHIO_KEYS (in module ogsSpy.tools.types), 224
MFP (class in ogsSpy.fileclasses.mfp), 108
MEKEYS (BC attribute), 44
MEKEYS (BlockFile attribute), 31
MEKEYS (CCT attribute), 51
MEKEYS (DDC attribute), 57
MEKEYS (FCT attribute), 63
MEKEYS (GEM attribute), 69
MEKEYS (IC attribute), 89
MEKEYS (KRC attribute), 100
MEKEYS (MCP attribute), 107
MEKEYS (MFP attribute), 114
MEKEYS (MMP attribute), 121
MEKEYS (MPD attribute), 128
MEKEYS (MSP attribute), 151
MEKEYS (NUM attribute), 159
MEKEYS (OUT attribute), 166
MEKEYS (PCS attribute), 173
MEKEYS (REI attribute), 185
MEKEYS (RFD attribute), 192
MEKEYS (ST attribute), 199
MEKEYS (TIM attribute), 205
MMP (class in ogsSpy.fileclasses.mmp), 115
MPD (class in ogsSpy.fileclasses.mpd), 123
MSH (class in ogsSpy.fileclasses.msh), 130
MSP (class in ogsSpy.fileclasses.msp), 146
MULTI_FILES (in module ogsSpy.tools.types), 224
MultiFile (class in ogsSpy.fileclasses.base), 35

N
name (ASC attribute), 38
name (BC attribute), 45
name (BlockFile attribute), 31
name (CCT attribute), 51
name (DDC attribute), 57
name (FCT attribute), 63
name (File attribute), 33
name (GEM attribute), 70
name (GEMinit attribute), 71
name (GLI attribute), 80
name (GLIext attribute), 81
name (IC attribute), 90
name (KRC attribute), 100
name (LineFile attribute), 35
name (MCP attribute), 107
name (MFP attribute), 114
name (MMP attribute), 122
name (MPD attribute), 129
name (MSH attribute), 139
name (MSP attribute), 152
name (NUM attribute), 159
name (OUT attribute), 166
name (PCS attribute), 173
name (PCT attribute), 175
name (PQC attribute), 177
name (PQCdata attribute), 179
name (REI attribute), 186
name (RFD attribute), 192
name (RF attribute), 92
name (ST attribute), 199
name (TIM attribute), 206
node_centroids (MSH attribute), 139
node_centroids_flat (MSH attribute), 140
NODE_NO (in module ogsSpy.tools.types), 224
NODE_NO (MSH attribute), 138
NODES (MSH attribute), 138
NUM (class in ogsSpy.fileclasses.num), 153

O
OGS (class in ogsSpy.ogs), 16
ogsSpy (module), 13
ogsSpy.fileclasses (module), 25
ogsSpy.fileclasses.asc (module), 36
ogsSpy.fileclasses.base (module), 26
ogsSpy.fileclasses.bc (module), 39
ogsSpy.fileclasses.cct (module), 46
ogsSpy.fileclasses.ddc (module), 52
ogsSpy.fileclasses.fct (module), 58
ogsSpy.fileclasses.gem (module), 64
ogsSpy.fileclasses.gli (module), 72
ogsSpy.fileclasses.gli.generator (module), 82
ogsSpy.fileclasses.ic (module), 84
R
radial() (in module ogs5py.fileclasses.gli.generator), 82
radial() (in module ogs5py.fileclasses.msh.generator), 144
read_file() (ASC method), 37
read_file() (BC method), 44
read_file() (BlockFile method), 30
read_file() (CCT method), 50
read_file() (DDC method), 56
read_file() (FCT method), 62
read_file() (File method), 32
read_file() (GEM method), 69
read_file() (GEMInit method), 71
read_file() (GLJ method), 76
read_file() (GLText method), 81
read_file() (IC method), 89
read_file() (KRC method), 99
read_file() (LineFile method), 34
read_file() (MCP method), 106
read_file() (MFP method), 113
read_file() (MMP method), 121
read_file() (MPD method), 128
read_file() (MSH method), 134
read_file() (MSP method), 151
read_file() (NUM method), 158
read_file() (OUT method), 165
read_file() (PCS method), 172
read_file() (PCT method), 175
read_file() (PQC method), 177
read_file() (PQCdat method), 179
read_file() (REI method), 185
read_file() (RFD method), 191
read_file() (RFR method), 92
read_file() (ST method), 198
read_file() (TIM method), 205
readpvd() (in module ogs5py.reader), 208
readpvd() (OGS method), 21
readpvd_single() (in module ogs5py.reader), 208
readtec_point() (in module ogs5py.reader), 208
readtec_point() (OGS method), 21
readtec_polyline() (in module ogs5py.reader), 209
readtec_polyline() (OGS method), 22
readvtk() (in module ogs5py.reader), 207
readvtk() (OGS method), 23
rectangular() (in module ogs5py.fileclasses.gli.generator), 82
rectangular() (in module ogs5py.fileclasses.msh.generator), 144
REI (class in ogs5py.fileclasses.rei), 180
remove_dim() (MSH method), 135
remove_point() (GLJ method), 77
remove_polyline() (GLJ method), 77
remove_surface() (GLJ method), 77
remove_volume() (GLJ method), 77
replace() (in module ogs5py.tools.tools), 215
reset() (ASC method), 37
reset() (BC method), 44
reset() (BlockFile method), 30
reset() (CCT method), 50
reset() (DDC method), 56
reset() (FCT method), 62
reset() (File method), 32
reset() (GEM method), 69
reset() (GEMInit method), 71
reset() (GLI method), 77
reset() (GLIext method), 81
reset() (IC method), 89
reset() (KRC method), 100
reset() (LineFile method), 34
reset() (MCP method), 106
reset() (MFP method), 113
reset() (MMP method), 121
reset() (MPD method), 128
reset() (MSH method), 135
reset() (MSP method), 151
reset() (NUM method), 158
reset() (OUT method), 165
reset() (PCS method), 172
reset() (PCT method), 175
reset() (PQC method), 177
reset() (PQCdat method), 179
reset() (REI method), 185
reset() (RFD method), 192
reset() (RFR method), 92
reset() (ST method), 198
reset() (TIM method), 205
reset_all() (MultiFile method), 35
reset_download() (in module ogs5py.tools.download), 227
RFD (class in ogs5py.fileclasses.rdf), 187
RFR (class in ogs5py.fileclasses.rfr), 90
rotate() (GLI method), 77
rotate() (MSH method), 135
rotate_points() (in module ogs5py.tools.tools), 215
rotation_matrix() (in module ogs5py.tools.tools), 216
run_model() (OGS method), 23
save() (ASC method), 37
save() (BC method), 44
save() (BlockFile method), 30
save() (CCT method), 50
save() (DDC method), 56
save() (FCT method), 63
save() (File method), 32
save() (GEM method), 69
save() (GEMInit method), 71
save() (GLI method), 77
save() (GLIext method), 81
save() (IC method), 89
save() (KRC method), 100
save() (LineFile method), 34
save() (MCP method), 106
save() (MFP method), 113
save() (MMP method), 121
save() (MPD method), 128
save() (MSH method), 135
save() (MSP method), 151
save() (NUM method), 158
save() (OUT method), 165
save() (PCS method), 172
save() (PCT method), 175
save() (PQC method), 177
save() (PQCdat method), 179
save() (REI method), 185
save() (RFD method), 192
save() (RFR method), 92
save() (ST method), 198
save() (TIM method), 205
search_mkey() (in module ogs5py.tools.tools), 216
search_task_id() (in module ogs5py.tools.tools), 216
set_dict() (GLI method), 77
set_dict() (MSH method), 135
set_material_id() (MSH method), 135
shift() (GLI method), 78
shift() (MSH method), 136
shift_points() (in module ogs5py.tools.tools), 216
show() (MSH method), 136
show_vtk() (in module ogs5py.tools.vtk_viewer), 229
S
save() (BC attribute), 44
save() (BlockFile attribute), 31
save() (CCT attribute), 51
save() (DDC attribute), 57
save() (FCT attribute), 63
save() (GEM attribute), 69
save() (IC attribute), 89
save() (KRC attribute), 100
save() (MCP attribute), 107
save() (MFP attribute), 114
save() (MMP attribute), 121
save() (MPD attribute), 128
save() (MSP attribute), 151
save() (NUM attribute), 159
save() (OUT attribute), 166
save() (PCS attribute), 173
save() (REI attribute), 185
save() (RFD attribute), 192
save() (ST attribute), 199
save() (TIM attribute), 205
specialrange() (in module ogs5py.tools.tools), 216
split_file_path() (in module ogs5py.tools.tools), 216
split_ply_path() (in module ogs5py.tools.output), 228
split_pnt_path() (in module ogs5py.tools.output), 228
SRF_KEY_LIST (in module ogs5py.types), 225
SRF_KEYS (in module ogs5py.types), 225
SRF_TYPES (in module ogs5py.types), 225
ST (class in ogs5py.fileclasses.st), 193
STD (BC attribute), 44
STD (BlockFile attribute), 31
STD (CCT attribute), 51
STD (DDC attribute), 57
STD (FCT attribute), 63
STD (GEM attribute), 69
STD (IC attribute), 89
STD (KRC attribute), 100
STD (MCP attribute), 107
STD (MFP attribute), 114
STD (MMP attribute), 121
STD (MPD attribute), 128
STD (MSP attribute), 152
STD (NUM attribute), 159
STD (OUT attribute), 166
STD (PCS attribute), 173
STD (REI attribute), 186
STD (RFD attribute), 192
STD (ST attribute), 199
STD (TIM attribute), 205
STR_TYPE (in module ogs5py.tools.types), 222
SURFACE_NAMES (GLI attribute), 79
SURFACE_NO (GLI attribute), 79
SURFACES (GLI attribute), 79
swap_axis() (GLI method), 78
swap_axis() (MSH method), 136

T

tab() (in module ogs5py.tools.script), 221
task_id (OGS attribute), 24
task_root (GEMinit attribute), 71
task_root (OGS attribute), 24
TIM (class in ogs5py.fileclasses.tim), 200
top_com (MPD attribute), 129
top_com (OGS attribute), 24
transform() (MSH method), 136
transform_points() (in module ogs5py.tools.tols), 217

U

uncomment() (in module ogs5py.tools.tools), 217
unique_rows() (in module ogs5py.tools.tools), 217
unique_rows_old() (in module ogs5py.tools.tools), 218
units (RFR attribute), 92
update_block() (BC method), 44
update_block() (BlockFile method), 30
update_block() (CCT method), 51
update_block() (DDC method), 57
update_block() (FCT method), 63
update_block() (GEM method), 69
update_block() (IC method), 89
update_block() (KRC method), 100
update_block() (MCP method), 106
update_block() (MFP method), 113
update_block() (MMP method), 121
update_block() (MPD method), 128
update_block() (MSP method), 151
update_block() (NUM method), 158
update_block() (OUT method), 165
update_block() (PCS method), 173
update_block() (REI method), 185
update_block() (RFD method), 192
update_block() (ST method), 198
update_block() (TIM method), 205

V

var_count (RFR attribute), 92
var_info (RFR attribute), 92
variables (RFR attribute), 92
VOL_KEY_LIST (in module ogs5py.types), 225
VOL_KEYS (in module ogs5py.types), 225
VOL_TYPES (in module ogs5py.types), 225
volume() (in module ogs5py.tools), 219
VOLUME_NAMES (GLI attribute), 79
VOLUME_NO (GLI attribute), 79
VOLUMES (GLI attribute), 79
volumes (MSH attribute), 140
volumes_flat (MSH attribute), 140
VTK_TYP (in module ogs5py.types), 225

W

write() (Output method), 213
write_file() (ASC method), 37
write_file() (BC method), 44
write_file() (BlockFile method), 31
write_file() (CCT method), 51
write_file() (DDC method), 57
write_file() (FCT method), 63
write_file() (File method), 32
write_file() (GEM method), 69
write_file() (GEMinit method), 71
write_file() (GLI method), 78
write_file() (GLExt method), 81
write_file() (IC method), 89
write_file() (KRC method), 100
write_file() (LineFile method), 34
write_file() (MCP method), 106
write_file() (MFP method), 113
write_file() (MMP method), 121
write_file() (MPD method), 128
write_file() (MSH method), 136
write_file() (MSP method), 151
write_file() (NUM method), 158
write_file() (OUT method), 166
write_file() (PCS method), 173
write_file() (PCT method), 175
write_file() (PQC method), 177
write_file() (PQCDat method), 179
write_file() (REI method), 185

242 Index
write_file() (RFD method), 192
write_file() (RFR method), 92
write_file() (ST method), 198
write_file() (TIM method), 205
write_input() (OGS method), 24