## Contents

1 Installation Guide
   1.1 What type of installation should I choose? .................................................. 1
   1.2 Using the Python-interpreter bundled to Thermo-Calc ....................................... 1
      1.2.1 Limitations .................................................................................................. 1
      1.2.2 Step 1: Install an IDE (Integrated Development Environment) ...................... 2
      1.2.3 Step 2: Configure PyCharm to use the bundled Python-interpreter ................. 2
      1.2.4 Step 3: Run a TC-Python Example ................................................................ 3
         1.2.4.1 Open the TC-Python Project in PyCharm ............................................... 3
   1.3 Installing TC-Python into the Python-interpreter of your choice ......................... 4
      1.3.1 Step 1: Install a Python Distribution ......................................................... 4
         1.3.1.1 Install Anaconda ................................................................................. 4
      1.3.2 Step 2: Install Thermo-Calc and the TC-Python SDK ................................... 4
      1.3.3 Step 3: Install TC-Python .......................................................................... 4
      1.3.4 Step 4: Install an IDE (Integrated Development Environment) ..................... 5
      1.3.5 Step 5: Open the IDE and Run a TC-Python Example ................................... 5
         1.3.5.1 Open the TC-Python Project in PyCharm ............................................... 5
         1.3.5.2 Fixing potential issues with the environment ........................................... 6
   1.4 Updating to a newer version .............................................................................. 6

2 Mac OS: Setting Environment Variables ......................................................... 9

3 Architecture overview ...................................................................................... 11
   3.1 TCPython .......................................................................................................... 11
   3.2 SystemBuilder and System .............................................................................. 12
   3.3 Calculation ........................................................................................................ 13
      3.3.1 Single equilibrium calculations ................................................................. 13
      3.3.2 Batch equilibrium calculations ................................................................. 14
      3.3.3 Precipitation calculations .......................................................................... 15
      3.3.4 Scheil calculations .................................................................................. 15
      3.3.5 Property diagram calculations .................................................................. 15
      3.3.6 Phase diagram calculations ...................................................................... 16
      3.3.7 Diffusion calculations ............................................................................... 17
      3.3.8 Property Model calculations .................................................................... 17
      3.3.9 Material to Material calculations .............................................................. 18
      3.3.10 Process Metallurgy calculations .............................................................. 18
   3.4 Result ................................................................................................................ 20
      3.4.1 DiffusionResult ......................................................................................... 21
   3.5 Property Model Framework ............................................................................. 21
      3.5.1 Property models vs. TC-Python .................................................................. 22
      3.5.2 Architecture ............................................................................................. 22
## 4 Best Practices

4.1 Re-use of the single equilibrium calculation state ........................................ 25
4.2 Re-use and saving of results ............................................................................. 26
4.3 All TC-Python objects are non-copyable .......................................................... 27
4.4 Python Virtual Environments ............................................................................ 27
4.5 Using *TCPython()* efficiently ........................................................................ 27
4.6 Parallel calculations .......................................................................................... 28
4.7 Handling crashes of the calculation engine ......................................................... 29
4.8 Using TC-Python within a Jupyter Notebook or the Python console ................. 30
4.9 Property Model Framework .............................................................................. 31
   4.9.1 Debugging Property Model code ................................................................. 31
   4.9.2 Developing Property Models in several files ................................................. 33
      4.9.2.1 side-by-side modules ......................................................................... 33
      4.9.2.2 common modules .................................................................................. 34
   4.9.3 Alternative Python for Property Models ................................................... 35
      4.9.3.1 Default bundled Python interpreter ...................................................... 35
      4.9.3.2 Configuring another Python interpreter ................................................. 35
4.10 Process Metallurgy Calculations ................................................................. 35
   4.10.1 Equilibrium calculations with changing elements between calculations .... 35
   4.10.2 Zones ........................................................................................................ 37
   4.10.3 Applications .............................................................................................. 37
   4.10.4 Implementation of practical process models .............................................. 37

## 5 API Reference

5.1 Calculations ..................................................................................................... 39
   5.1.1 Module “single_equilibrium” ..................................................................... 39
   5.1.2 Module “batch_equilibrium” ...................................................................... 52
   5.1.3 Module “precipitation” .............................................................................. 57
   5.1.4 Module “scheil” ......................................................................................... 79
   5.1.5 Module “step_or_map_diagrams” ................................................................. 89
   5.1.6 Module “diffusion” .................................................................................... 117
   5.1.7 Module “propertymodel” .......................................................................... 155
   5.1.8 Module “material_to_material” ................................................................. 160
   5.1.9 Package “process_metallurgy” ................................................................... 183
      5.1.9.1 Module “base” ..................................................................................... 183
      5.1.9.2 Module “equilibrium” ......................................................................... 188
      5.1.9.3 Module “process” ................................................................................ 198
   5.2 Module “system” ........................................................................................... 216
   5.3 Module “entities” ........................................................................................... 224
   5.4 Module “server” ............................................................................................. 228
   5.5 Module “quantity_factory” ............................................................................ 234
   5.6 Module “utils” ................................................................................................ 246
   5.7 Module “propertymodel_sdk” ......................................................................... 248
   5.8 Module “exceptions” ...................................................................................... 265
   5.9 Module “abstract_base” ................................................................................ 267

## 6 Troubleshooting

6.1 Diagnostics script ............................................................................................ 275
6.2 “No module named tc_python” error on first usage ........................................... 275
6.3 “pip install” fails with “Failed to establish a new network connection” or similar 278

Python Module Index ........................................................................................................ 279
CHAPTER ONE

INSTALLATION GUIDE

This guide helps you to get a working TC-Python API installation.

There is a PDF guide included with your installation. In the Thermo-Calc menu, select Help → Manuals Folder. Then double-click to open the Software Development Kits (SDKs) folder.

Note: A license is required to run TC-Python.

1.1 What type of installation should I choose?

There are two possibilities to install TC-Python:

1. Using the Python-interpreter bundled to Thermo-Calc: This interpreter has TC-Python preinstalled together with some popular Python-packages. This is the recommended option for new users to TC-Python, but it is limited to the preinstalled packages.

2. Installing TC-Python into the Python-interpreter of your choice: This is the recommended option for any more advanced usage and provides full flexibility.

1.2 Using the Python-interpreter bundled to Thermo-Calc

Note: A Python-interpreter is bundled to Thermo-Calc beginning with version 2021a.

1.2.1 Limitations

The bundled Python 3.7.2 interpreter is containing the following major packages:

<table>
<thead>
<tr>
<th>Package</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>matplotlib</td>
<td>3.3.2</td>
</tr>
<tr>
<td>numpy</td>
<td>1.19.2</td>
</tr>
<tr>
<td>scikit-learn</td>
<td>0.23.2</td>
</tr>
<tr>
<td>scipy</td>
<td>1.5.2</td>
</tr>
<tr>
<td>TC-Python</td>
<td>2022b</td>
</tr>
</tbody>
</table>
Please contact the Thermo-Calc support if you think that further packages might be useful in future releases.

Note: The following TC-Python examples are requiring additional packages that are not available in the bundled Python-interpreter, they can therefore not be run:

- `pyex_M_01_Input_from_file.py`
- `pyex_M_02_Output_to_file.py`

Warning: The Python-interpreter bundled to Thermo-Calc is also used for running the property models in Thermo-Calc. Any changes to the interpreter packages can therefore break Thermo-Calc and should be avoided. If the installation gets broken, it can be fixed by reinstalling Thermo-Calc after having removed it.

### 1.2.2 Step 1: Install an IDE (Integrated Development Environment)

Any editor can be used to write the Python code, but an IDE is recommended, e.g. PyCharm. These instructions are based on the use of PyCharm.

Use of an IDE will give you access to code completion, which is of great help when you use the API as it will give you the available methods on the objects you are working with.

2. Click to choose your OS and then click Download. You can use the Community version of PyCharm.
3. Follow the instructions. It is recommended you keep all the defaults.

Note: For Mac installations, you also need to set some environment variables as described below in *Mac OS: Setting Environment Variables*.

### 1.2.3 Step 2: Configure PyCharm to use the bundled Python-interpreter

Open PyCharm and configure the interpreter:

1. Go the menu File → Settings.
2. Navigate in the tree to Project: YourProjectName and choose Project Interpreter.
3. Click on the settings symbol close to the Project Interpreter dropdown menu and choose Add.
4. Now choose System Interpreter and add the bundled Thermo-Calc Python 3 interpreter. It is located in different places depending on the operating system:
<table>
<thead>
<tr>
<th>Operating system</th>
<th>Path to the bundled Python-interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>C:\Program Files\Thermo-Calc\2022b\python\python.exe</td>
</tr>
<tr>
<td>Linux</td>
<td>/home/UserName/Thermo-Calc/2022b/python/bin/python3</td>
</tr>
<tr>
<td>MacOS</td>
<td>/Applications/Thermo-Calc-2022b.app/Contents/Resources/python/bin/python3</td>
</tr>
</tbody>
</table>

5. Select your added interpreter and confirm.

### 1.2.4 Step 3: Run a TC-Python Example

Now you are ready to start working with TC-Python.

It is recommended that you open one or more of the included examples to both check that the installation has worked and to start familiarizing yourself with the code.

#### 1.2.4.1 Open the TC-Python Project in PyCharm

When you first open the TC-Python project and examples, it can take a few moments for the Pycharm IDE to index before some of the options are available.

1. Open PyCharm and then choose **File** → **Open**. The first time you open the project you will need to navigate to the path of the TC-Python installation:

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Path to the TC-Python folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>C:\Users\UserName\Documents\Thermo-Calc\2022b\SDK\TC-Python</td>
</tr>
<tr>
<td>Linux</td>
<td>/home/UserName/Thermo-Calc/2022b/SDK/TC-Python</td>
</tr>
<tr>
<td>MacOS</td>
<td>/Users/Shared/Thermo-Calc/2022b/SDK/TC-Python</td>
</tr>
</tbody>
</table>

2. Click on the **Examples** folder and then click **OK**.

3. From any subfolder:
   - Double-click to open an example file to examine the code.
   - Right-click an example and choose **Run**.

**Note:** If you are not following the recommended approach and create a new project (**File** → **New Project**...), you need to consider that by default the options to choose the interpreter are hidden within the **Create Project** window. So click on **Project Interpreter: New Virtual Environment** and in most cases choose your **System Interpreter** containing the Python bundled to Thermo-Calc instead of the default **New Virtual Environment**.
1.3 Installing TC-Python into the Python-interpreter of your choice

1.3.1 Step 1: Install a Python Distribution

If you already have a Python distribution installation, version 3.5 or higher, skip this step.

These instructions are based on using the Anaconda platform for the Python distribution. Install version 3.5 or higher to be able to work with TC-Python, although it is recommended that you use the most recent version.

1.3.1.1 Install Anaconda

2. Click to choose your OS (operating system) and then click Download. Follow the instructions. It is recommended you keep all the defaults.

1.3.2 Step 2: Install Thermo-Calc and the TC-Python SDK

Note: TC-Python is available starting with Thermo-Calc version 2018a.

1. Install Thermo-Calc
2. When the installation is complete, open the TC-Python folder that includes the *.whl file needed for the next step. There is also an file:Examples folder with Python files you can use in the IDE to understand and work with TC-Python.

1.3.3 Step 3: Install TC-Python

On Windows, it is recommended that you use the Python distribution prompt (i.e. Anaconda, . . . ), especially if you have other Python installations. Do not use Virtual Environments unless you have a good reason for that.

1. Open the command line. For example, in Anaconda on a Windows OS, go to Start→Anaconda→Anaconda Prompt.
2. At the command line, enter the following. Make sure there are no spaces at the end of the string or in the folder name or it will not run:

   pip install <path to the TC-Python folder>/TC_Python-<version>-py3-none-any.whl

   ![](https://via.placeholder.com/150)

   Tip: Note that on Linux depending on the interpreter usually pip3 is used.

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Path to the TC-Python folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>C:\Users\UserName\Documents\Thermo-Calc\2022b\SDK\TC-Python</td>
</tr>
<tr>
<td>Linux</td>
<td>/home/UserName/Thermo-Calc/2022b/SDK/TC-Python</td>
</tr>
<tr>
<td>MacOS</td>
<td>/Users/Shared/Thermo-Calc/2022b/SDK/TC-Python</td>
</tr>
</tbody>
</table>
3. Press <Enter>. When the process is completed, there is a confirmation that TC-Python is installed.

**Note:** If your computer is located behind a proxy-server, the default `pip`-command will fail with a network connection error. In that case you need to install the dependencies of TC-Python in a special configuration:

```
 pip install -proxy user:password@proxy_ip:port py4j jproperties
```

See “`pip install` fails with “Failed to establish a new network connection” or similar” for detailed information.

### 1.3.4 Step 4: Install an IDE (Integrated Development Environment)

Any editor can be used to write the Python code, but an IDE is recommended, e.g. PyCharm. These instructions are based on the use of PyCharm.

Use of an IDE will give you access to code completion, which is of great help when you use the API as it will give you the available methods on the objects you are working with.

2. Click to choose your OS and then click **Download**. You can use the **Community** version of PyCharm.
3. Follow the instructions. It is recommended you keep all the defaults.

**Note:** For Mac installations, you also need to set some environment variables as described below in *Mac OS: Setting Environment Variables*.

### 1.3.5 Step 5: Open the IDE and Run a TC-Python Example

After you complete all the software installations, you are ready to open the IDE to start working with TC-Python.

It is recommended that you open one or more of the included examples to both check that the installation has worked and to start familiarizing yourself with the code.

#### 1.3.5.1 Open the TC-Python Project in PyCharm

When you first open the TC-Python project and examples, it can take a few moments for the Pycharm IDE to index before some of the options are available.

1. Open PyCharm and then choose **File→Open**. The first time you open the project you will need to navigate to the path of the TC-Python installation.

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Path to the TC-Python folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>C:\Users\UserName\Documents\Thermo-Calc\2022b\SDK\TC-Python</td>
</tr>
<tr>
<td>Linux</td>
<td>/home/UserName/Thermo-Calc/2022b/SDK/TC-Python</td>
</tr>
<tr>
<td>MacOS</td>
<td>/Users/Shared/Thermo-Calc/2022b/SDK/TC-Python</td>
</tr>
</tbody>
</table>

1.3. Installing TC-Python into the Python-interpreter of your choice
2. Click on the Examples folder and then click OK.
3. From any subfolder:
   - Double-click to open an example file to examine the code.
   - Right-click an example and choose Run.

### 1.3.5.2 Fixing potential issues with the environment

In most cases you should run TC-Python within your global Python 3 interpreter and not use Virtual Environments unless you have a good reason to do so. A common problem on first usage of TC-Python is the error message “No module named tc_python”. You can resolve this and other problems with the interpreter settings as follows:

1. Go to the menu File → Settings.
2. Navigate in the tree to Project: YourProjectName and choose Project Interpreter.
3. Click on the settings symbol close to the Project Interpreter dropdown menu and choose Add.
4. Now choose System Interpreter and add your existing Python 3 interpreter.
5. Select your added interpreter and confirm.

**Note:** If you are not following the recommended approach and create a new project (File → New Project…), you need to consider that by default the options to choose the interpreter are hidden within the Create Project window. So click on Project Interpreter: New Virtual Environment and in most cases choose your System Interpreter instead of the default New Virtual Environment.

**Note:** If you really need to use a Virtual Environment, please consider the hints given in the Python Virtual Environments chapter.

### 1.4 Updating to a newer version

When updating to a newer version of Thermo-Calc, you need to also install the latest version of TC-Python. This is not necessary if you are using the bundled Python-interpreter that has it automatically installed. It is not sufficient to run the installer of Thermo-Calc:

```bash
pip install <path to the TC-Python folder>/TC_Python-<version>-py3-none-any.whl
```

**Tip:** Note that on Linux depending on the interpreter usually pip3 is used.

In case of problems you may wish to uninstall the previous version of TC-Python in advance:

```bash
pip uninstall TC-Python
pip install <path to the TC-Python folder>/TC_Python-<version>-py3-none-any.whl
```
However, that should normally not be required.

You can check the currently installed version of TC-Python by running:

```
pip show TC-Python
```
MAC OS: SETTING ENVIRONMENT VARIABLES

In order to use TC-Python on Mac you need to set some environment variables.

TC22B_HOME=/Applications/Thermo-Calc-2022b.app/Contents/Resources

If you use a license server:

LSHOST=<name-of-the-license-server>

If you have a node-locked license:

LSHOST=NO-NET
LSERVRC=/Users/Shared/Thermo-Calc/lservrc

In PyCharm, you can add environment variables in the configurations.

Select Run → Edit Configurations to open the Run/Debug Configurations window. Choose Templates and then Python. Enter the environment variable(s) by clicking the button to the right of the Environment Variables text field. Now the environment variables(s) will be set for each new configuration by default.

Note: Existing configurations need to be removed and recreated to obtain the environment variables in them.

The same way for configuring the environment variables can be used on other operating systems as if necessary.
CHAPTER
THREE

ARCHITECTURE OVERVIEW

TC-Python contains classes of these types:

- **TCPython** – this is where you start with general settings.
- **SystemBuilder** and **System** – where you choose database and elements etc.
- **Calculation** – where you choose and configure the calculation.
- **Result** – where you get the results from a calculation you have run.

### 3.1 TCPython

This is the starting point for all TC-Python usage.

You can think of this as the start of a “wizard”.

You use it to select databases and elements. That will take you to the next step in the wizard, where you configure the system.

**Example:**

```python
from tc_python import *

with TCPython() as start:
    start.select_database_and_elements(...
    # e.t.c
    # after with clause

# or like this
with TCPython():
    SetUp().select_database_and_elements(...
    # e.t.c
    # after with clause
```

**Tip:** If you use TC-Python from Jupyter Lab / Notebook, you should use TC-Python slightly different to be able to use multiple cells. See *Using TC-Python within a Jupyter Notebook or the Python console* for details.

**Note:** When your python script runs a row like this:

```python
with TCPython() as start:
```
a process running a calculation server starts. Your code, via TC-Python, uses socket communication to send and receive messages to and from that server.

When your Python script has run as far as this row:

```python
# after with clause
```

the calculation server automatically shuts down, and all temporary files are deleted. It is important to ensure that this happens by structuring your Python code using a `with()` clause as in the above example.

**Note:** To re-use results from previous calculations, set a folder where TC-Python saves results, and looks for previous results.

This is done with the function `set_cache_folder()`.

```python
from tc_python import *
with TCPython() as start:
    start.set_cache_folder("cache")
```

This folder can be a network folder and shared by many users. If a previous TC-Python calculation has run with the same cache_folder and EXACTLY the same system and calculation settings, the calculation is not re-run. Instead the result is automatically loaded from disk.

It is also possible to explicitly save and load results.

```python
from tc_python import *
with TCPython() as start:
    #... diffusion calculation (could be any calculation type)
    calculation_result.save_to_disk('path to folder')
    #...
    loaded_result = start.load_result_from_disk().diffusion('path to folder')
```

### 3.2 SystemBuilder and System

A **SystemBuilder** is returned when you have selected your database and elements in **TCPython**.

The **SystemBuilder** lets you further specify your system, for example the phases that should be part of your system.

**Example:**

```python
from tc_python import *
with TCPython() as start:
    start.select_database_and_elements("ALDEMO", ["Al", "Sc"])
    # e.t.c
```

When all configuration is done, you call `get_system()` which returns an instance of a **System** class. The **System** class is fixed and cannot be changed. If you later want to change the database, elements or something else, change the **SystemBuilder** and call `get_system()` again, or create a new **SystemBuilder** and call `get_system()`.

From the **System** you can create one or more calculations, which is the next step in the “wizard”.
Note: You can use the same System object to create several calculations.

### 3.3 Calculation

The best way to see how a calculation can be used is in the TC-Python examples included with the Thermo-Calc installation.

Some calculations have many settings. Default values are used where it is applicable, and are overridden if you specify something different.

When you have configured your calculation you call calculate() to start the actual calculation. That returns a Result, which is the next step.

#### 3.3.1 Single equilibrium calculations

In single equilibrium calculations you need to specify the correct number of conditions, depending on how many elements your System contains.

You do that by calling set_condition().

An important difference from other calculations is that single equilibrium calculations have two functions to get result values.

The calculate() method, which gives a SingleEquilibriumTempResult, is used to get actual values. This result is “temporary”, meaning that if you run other calculations or rerun the current one, the resulting object no longer gives values corresponding to the first calculation.

This is different from how other calculations work. If you want a Result that you can use after running other calculations, you need to call calculate_with_state(), which returns a SingleEquilibriumResult.

Note: calculate() is the recommended function and works in almost all situations. Also it has much better performance than calculate_with_state().

**Example:**

```python
from tc_python import *

with TCPython() as start:
    gibbs_energy = {
        start.
        select_database_and_elements("FEDEMO", ["Fe", "Cr", "C"]).
        get_system().
        with_single_equilibrium_calculation().
        set_condition(ThermodynamicQuantity.temperature(), 2000.0).
        set_condition(ThermodynamicQuantity.mole_fraction_of_a_component("Cr ↪"), 0.1).
        set_condition(ThermodynamicQuantity.mole_fraction_of_a_component("C"), 0.01).
        calculate().
        get_value_of("G")
    }
```

3.3. Calculation 13
3.3.2 Batch equilibrium calculations

Batch equilibrium calculations are used when you want to do many single equilibrium calculations and it is known from the beginning which result values are required from the equilibrium. This is a vectorized type of calculation that can reduce the overhead from Python and TC-Python similar to the approach used in *numpy*-functions for example.

**Tip:** The performance of batch equilibrium calculations can be significantly better than looping and using single equilibrium calculations if the actual Thermo-Calc calculation is fast. There is little advantage if the Thermo-Calc equilibrium calculations take a long time (typically for large systems and databases).

Example:

```python
from tc_python import *

with TCPython() as start:
    calculation = start
        .set_cache_folder(os.path.basename(__file__) + "_cache")
        .select_database_and_elements("NIDEMO", ["Ni", "Al", "Cr"])
        .get_system()
        .with_batch_equilibrium_calculation()
        .set_condition("T", 800.1)
        .set_condition("X(Al)", 1E-2)
        .set_condition("X(Cr)", 1E-2)
        .disable_global_minimization()

list_of_x_Al = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list_of_x_Cr = [3, 5, 7, 9, 11, 13, 15]
lists_of_conditions = []
for x_Al in list_of_x_Al:
    for x_Cr in list_of_x_Cr:
        lists_of_conditions.append(["X(Al)", x_Al / 100],
                                   ["X(Cr)", x_Cr / 100])
    calculation.set_conditions_for_equilibria(lists_of_conditions)

results = calculation.calculate(["BM", "VM"])

masses = results.get_values_of("BM")
volumes = results.get_values_of("VM")

print(masses)
print(volumes)
```
3.3.3 Precipitation calculations

All that can be configured in the Precipitation Calculator in Graphical Mode can also be done here in this calculation. However, you must at least enter a matrix phase, a precipitate phase, temperature, simulation time and compositions.

**Example:**

```python
from tc_python import *

with TCPython() as start:
    precipitation_curve = (start.
        select_thermodynamic_and_kinetic_databases_with_elements("ALDEMO", "MALDEMO", ["Al", "Sc"]).
        get_system().
        with_isothermal_precipitation_calculation().
            set_composition("Sc", 0.18).
            set_temperature(623.15).
            set_simulation_time(1e5).
            with_matrix_phase(MatrixPhase("FCC_A1").
                add_precipitate_phase(PrecipitatePhase("AL3SC"))).
        calculate()
    )
```

3.3.4 Scheil calculations

All Scheil calculations available in Graphical Mode or Console Mode can also be done here in this calculation. The minimum you need to specify are the elements and compositions. Everything else is set to a default value.

**Example:**

```python
from tc_python import *

with TCPython() as start:
    temperature_vs_mole_fraction_of_solid = (start.
        select_database_and_elements("FEDEMO", ["Fe", "C"]).
        get_system().
        with_scheil_calculation().
            set_composition("C", 0.3).
            calculate().
            get_values_of(ScheilQuantity.temperature(),
                ScheilQuantity.mole_fraction_of_all_solid_phases())
    )
```

3.3.5 Property diagram calculations

For the property diagram (step) calculation, everything that you can configure in the Equilibrium Calculator when choosing One axis in Graphical Mode can also be configured in this calculation. In Console Mode the property diagram is created using the Step command. The minimum you need to specify are elements, conditions and the calculation axis. Everything else is set to default values, if you do not specify otherwise.

**Example:**

```python

```

3.3. Calculation
from tc_python import *

with TCPython() as start:
    property_diagram = {
        start.
        select_database_and_elements("FEDEMO", ["Fe", "C"]).
        get_system().
        with_property_diagram_calculation().
            with_axis(CalculationAxis(ThermodynamicQuantity.temperature()).
                set_min(500).
                set_max(3000)).
                set_condition(ThermodynamicQuantity.mole_fraction_of_a_component("C"),
                → 0.01).
                calculate().
                get_values_grouped_by_stable_phases_of(ThermodynamicQuantity.
                →temperature(),
                    ThermodynamicQuantity.volume_)
        →fraction_of_a_phase("ALL")
    }

3.3.6 Phase diagram calculations

For the phase diagram (map) calculation, everything that you can configure in the Equilibrium Calculator when choosing Phase diagram in Graphical Mode can also be configured in this calculation. In Console Mode the phase diagram is created using the Map command. The minimum you need to specify are elements, conditions and two calculation axes. Everything else is set to default values, if you do not specify otherwise.

Example:

from tc_python import *

with TCPython() as start:
    phase_diagram = {
        start.
        select_database_and_elements("FEDEMO", ["Fe", "C"]).
        get_system().
        with_phase_diagram_calculation().
            with_first_axis(CalculationAxis(ThermodynamicQuantity.temperature()).
                set_min(500).
                set_max(3000)).
                with_second_axis(CalculationAxis(ThermodynamicQuantity.mole_fraction_of_a_component("C")),
                → set_min(0).
                → set_max(1)).
                set_condition(ThermodynamicQuantity.mole_fraction_of_a_component("C"),
                → 0.01).
                calculate().
                get_values_grouped_by_stable_phases_of(ThermodynamicQuantity.mass_)
                → fraction_of_a_component("C"),
            → temperature())
        }

16 Chapter 3. Architecture overview
3.3.7 Diffusion calculations

For diffusion calculations, everything that you can configure in the Diffusion Calculator can also be configured in this calculation. The minimum you need to specify are elements, temperature, simulation time, a region with a grid and width, a phase and an initial composition.

Example:

```python
from tc_python import *
with TCPython() as start:
    diffusion_result = {
        start.
        select_thermodynamic_and_kinetic_databases_with_elements("FEDEMO", "MFEDEMO", ["Fe", "Ni"]).
        get_system().
        with_isothermal_diffusion_calculation().
        set_temperature(1400.0).
        set_simulation_time(108000.0).
        add_region(Region("Austenite").set_width(1E-4).
            with_grid(CalculatedGrid.linear().set_no_of_points(50)).
            with_composition_profile(CompositionProfile().
                add("Ni", ElementProfile.linear(10.0, 50.0))
            ).
            add_phase("FCC_A1")).
        calculate()
    }
    distance, ni_fraction = diffusion_result.get_mass_fraction_of_component_at_time("Ni", 108000.0)
```

3.3.8 Property Model calculations

For Property Model calculations, everything that you can configure in the Property Model Calculator in Graphical Mode can also be configured in this calculation. The minimum you need to specify are elements, composition and which Property Model you want to use.

Example:

```python
from tc_python import *
with TCPython() as start:
    print("Available Property Models: "/format(start.get_property_models()))
    property_model = {
        start.
        select_database_and_elements("FEDEMO", ["Fe", "C"]).
        get_system().
        with_property_model_calculation("Driving force").
        set_composition("C", 1.0).
        set_argument("precipitate", "GRAPHITE")
        print("Available arguments: "/format(property_model.get_arguments()))
        result = property_model.calculate()
        print("Available result quantities: "/format(result.get_result_quantities()))
        driving_force = result.get_value_of("normalizedDrivingForce")
```
3.3.9 Material to Material calculations

Material to Material calculations are generally regular single equilibrium, property diagram or phase diagram calculations but they are specialised to handle the mixture of two materials A and B. Everything that you can configure in the Material to Material Calculator in Graphical Mode can also be configured in this calculation. The minimum required configuration is shown below for a Property diagram calculation for varying amount of material B. The other calculators (single fraction of material B and phase diagram calculations) are configured in a similar way.

Example:

```python
from tc_python import *

with TCPython() as start:
    material_to_material_property_diagram = (start.
        select_database_and_elements("FEDEMO", ["Fe", "Cr", "Ni", "C"]).
        get_system().
        with_material_to_material().
        with_property_diagram_calculation().
        set_material_a({"Cr": 10.0, "Ni": 15.0}, "Fe").
        set_material_b({"Cr": 15.0, "Ni": 10.0}, "Fe").
        set_activities({"C": 0.1}).
        with_constant_condition(ConstantCondition.temperature(800 + 273.15)).
        with_axis(MaterialToMaterialCalculationAxis.fraction_of_material_b(from_fraction=0.0,
                                                                         to_fraction=1.0,
                                                                         start_fraction=0.5))
    )
    result = material_to_material_property_diagram.calculate()
    data = result.get_values_grouped_by_quantity_of(MATERIAL_B_FRACTION,
                                                   ThermodynamicQuantity.volume_fraction_of_a_phase(ALL_PHASES))
    for group in data.values():
        fractions_of_b = group.x
        volume_fractions_of_phase = group.y
        phase_name = group.label
```

3.3.10 Process Metallurgy calculations

Process Metallurgy calculations are specialized to support the convenient handling of component-based additions (i.e., slag compositions such as 50% Al₂O₃ - 30% CaO - 20% SiO₂), provide tailor-made result quantities, a framework for developing kinetic process simulations, and more useful features.

There are two distinct types of calculations:

- `tc_python.process_metallurgy.equilibrium.EquilibriumCalculation`: isothermal and adiabatic equilibrium calculations
- `tc_python.process_metallurgy.process.ProcessSimulationCalculation`: a kinetic process simulation framework, based on an Effective Equilibrium Reaction Zone (EERZ) approach

Equilibrium calculation example:
Equilibrium calculations are useful in a large range of situations when considering the kinetics of a process is unnecessary.

```python
from tc_python import *

with TCPython() as session:
    metal = EquilibriumAddition(#{"Fe": None, "C": 4.5, "Si": 1.0}, 100e3, temperature=1650 + 273.15)
    slag = EquilibriumAddition(#{"CaO": 75, "Al2O3": 25}, 3e3, temperature=1600 + 273.15)
    gas = EquilibriumGasAddition(#{"O2": 100}, 1000, amount_unit=GasAmountUnit.NORM_CUBIC_METER)
    calc = session.with_metallurgy().with_adiabatic_equilibrium_calculation(ProcessDatabase.OXDEMO)
    (calc
     .add_addition(metal)
     .add_addition(slag)
     .add_addition(gas))
    result = calc.calculate()
    print(f"Stable phases: {result.get_stable_phases()}, temperature: {result.get_temperature()} K")
```

**Process simulation example:**

TC-Python is providing a framework for modelling in principle any process in metallurgy, especially steel-making. It is up to the user to actually develop a concrete model for the process in question. The framework is in the current release limited to one reaction zone connecting two bulk zones. These bulk zones are typically the steel melt and the top slag, but not limited to that. The framework in its current version has proven to be useful to model industrial ladle furnaces, AOD- and VOD-converters and more. Process features such as heating and cooling, heat transfer between the bulk zones, inclusion formation and their flotation, etc., can be modelled.

This is a very simplified minimal but complete model mimicking a BOF process:

```python
from tc_python import *

with TCPython() as session:
    calc = (session.with_metallurgy()
        .with_adiabatic_process_calculation(ProcessDatabase.OXDEMO)
        .set_end_time(15 * 60))
    steel_zone = MetalBulkZone(density=7800)
    slag_zone = SlagBulkZone(density=4500)
    steel_zone.add_addition(SingleTimeAddition(#{"Fe": None, "C": 4.5, "Si": 1.0}, 120e3,
    temperature=1600 + 273.15), time=0)
    slag_zone.add_addition(SingleTimeAddition(#{"CaO": 75, "SiO2": 25}, 1.2e3,
    temperature=1500 + 273.15,
    composition_unit=CompositionUnit.MOLE_PERCENT), time=0)
    steel_zone.add_continuous_addition(ContinuousGasAddition(#{"O2": 100}, 1,
    rate_unit=GasRateUnit.NORM_CUBIC_METER_PER_SEC))
```

(continues on next page)
```python
calc.with_reaction_zone(ReactionZone(area=10.0,
    left_zone=steel_zone, mass_transfer_
    coefficient_left=1.0e-5,
    right_zone=slag_zone, mass_transfer_
    coefficient_right=1.0e-6))

result = calc.calculate()

print(f"Stable phases in the steel melt: {result.get_stable_phases(steel_zone)}")
print(f"C-content in steel vs. time: {result.get_composition_of_phase_group(steel_ zone, PhaseGroup.ALL_METAL)['C']}")
```

### 3.4 Result

All calculations have a method called `calculate()` that starts the calculations and when finished, returns a `Result`. The `Result` classes have very different methods, depending on the type of calculation. The `Result` is used to get numerical values from a calculation that has run. The `Result` can be saved to disk by the method `save_to_disk()`. Previously saved results can be loaded by the method `load_result_from_disk()` on the `SetUp` class.

**Example:**

```python
# code above sets up the calculation
r = calculation.calculate()
time, meanRadius = r.get_mean_radius_of("AL3SC")
```

The `Result` objects are completely independent from calculations done before or after they are created. The objects return valid values corresponding to the calculation they were created from, for their lifetime. The only exception is if you call `calculate()` and not `calculate_with_state()` on a single equilibrium calculation. As in the following example you can mix different calculations and results, and use old results after another calculation has run.

**Example:**

```python
# ...
# some code to set up a single equilibrium calculation
# ...

single_eq_result = single_eq_calculation.calculate_with_state()

# ...
# some code to set up a precipitation calculation
# ...

prec_result = precipitation_calculation.calculate()

# ...
# some code to set up a Scheil calculation
# ...
```
scheil_result = scheil_calculations.calculate()

# now it is possible to get results from the single equilibrium calculation,
# without having to re-run it (because it has been calculated with saving of the
# state)

gibbs = single_eq_result.get_value_of("G")

### 3.4.1 DiffusionResult

The DiffusionResult class, that is returned when calling calculate() on any DiffusionCalculation, has the possibility to create a ContinuedDiffusionCalculation, in addition to the “normal” functionality for results. This makes it possible to run a diffusion calculation and then, depending on the result, change some settings and continue.

Example:

```python
# ...
# some code to set up a Diffusion calculation
# ...
first_diffusion_result = diffusion_calculation.calculate()

continued_calculation = first_diffusion_result.with_continued_calculation()

continued_calculation.set_simulation_time(110000.0)
continued_calculation.with_left_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element('C', 1.0))
second_result = continued_calculation.calculate()
# ...
# Now you can use second_result to get calculated values, just as normal.
# You can also use first_diffusion_result even after second_result is created.
# You can also use second_result (and even first_diffusion_result) to create a new
# ContinuedDiffusionCalculation by calling with_continued_calculation.
```

### 3.5 Property Model Framework

The Python Property Model SDK extends the Thermo-Calc software to enable you to create your own Property Models. A Property Model is a Python-based calculation that can use any TC-Python functionality (including diffusion and precipitation calculations) but is usable through the Graphical User Interface (UI) of Thermo-Calc in a more simple way. It is typically used to model material properties but by no means limited to that. Examples of Property Models provided by Thermo-Calc include Martensite and Pearlite formation in steel.

The Property Model Framework uses standard Python 3 beginning with Thermo-Calc 2021a and can access all TC-Python functionality and any Python package including numpy, scipy, tensorflow, etc. The actual calculation code is nearly identical, regardless if called from within a Property Model or from standard Python.

This is a complete rewrite of the original version of the framework that was based on Jython 2.7 and therefore had a number of limitations. **Property models written with the old Property Model Framework before Thermo-Calc 2021a are not compatible with the new framework.** However, the migration should be relatively easy because the syntax was changed as little as possible.
3.5.1 Property models vs. TC-Python

The main difference between a Property Model and regular TC-Python code is that a Property Model is directly integrated into the UI of Thermo-Calc via a plugin architecture while TC-Python code can only be accessed by programs and scripts written in Python.

The user should develop a Property Model if the functionality needs to be available from the Thermo-Calc UI, especially if it should be applied by other users not familiar to programming languages. Otherwise it is preferable to implement the functionality directly in a TC-Python program. If required, Property Models can as well be accessed from within TC-Python.

3.5.2 Architecture

Every Property Model needs to contain a class that implements the interface tc_python.propertymodel_sdk.PropertyModel. There are naming conventions that must to be fulfilled: the file name is required to follow the the pattern XYPythonModel.py and the name of the class needs to match this. Additionally the file must be placed in a directory named XYPython within the Property Model directory. The content of the placeholder XY can be freely chosen.

A simple complete Property Model, saved in a file called SimplePythonModel.py in the directory SimplePython, looks like this:

```python
from tc_python import *

class SimplePythonModel(PropertyModel):
    def provide_model_category(self) -> List[str]:
        return "Demo"

    def provide_model_name(self) -> str:
        return "My Demo Model"

    def provide_model_description(self) -> str:
        return "This is a demo model."

    def provide_ui_panel_components(self) -> List[UIComponent]:
        return [UIBooleanComponent("CHECKBOX", "Should this be checked?", "Simple checkbox", setting=False)]

    def provide_calculation_result_quantities(self) -> List[ResultQuantity]:
        return [create_general_quantity("RESULT", "A result")]

    def evaluate_model(self, context: CalculationContext):
        if context.get_ui_boolean_value("CHECKBOX"):
            self.logger.info("The checkbox is checked")
            # obtain the entered values from the GUI
            composition_as_mass_fraction = context.get_mass_fractions()
            temp_in_k = context.get_temperature()
            calc = context.system.with_single_equilibrium_calculation()
            # continue with a TC-Python calculation now ...
            context.set_result_quantity_value("RESULT", 5.0)  # the value would normally have been calculated
```

The basic building blocks of the Property Model API are:
• `tc_python.propertymodel_sdk.ResultQuantity`: Defines a calculation result of a Property Model that will be provided to the UI after each model evaluation

• `tc_python.propertymodel_sdk.CalculationContext`: Provides access to the data from the UI (such as the entered composition and temperature) and to the current TC-Python system object which is the entrypoint for using TC-Python from within the Property Model

• `tc_python.propertymodel_sdk.UIComponent`: These are the UI-components that create the user interface of the Property Model within the model panel of the Thermo-Calc application UI. Different components are available (for example checkboxes, text fields and lists).

### 3.5.3 Property Model directory

The Property Model `py`-files need to be located within subdirectories of the *Property Model directory*, e.g. `PropertyModelDir/XYPython/XYPythonModel.py`. The default Property Model directory can be changed in the menu `Tools -> Options` in the graphical user interface.

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Default Property Model directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td><code>C:\Users\UserName\Documents\Thermo-Calc\2022b\PropertyModels</code></td>
</tr>
<tr>
<td>Linux</td>
<td><code>/home/UserName/Thermo-Calc/2022b/PropertyModels</code></td>
</tr>
<tr>
<td>MacOS</td>
<td><code>/Users/Shared/Thermo-Calc/2022b/PropertyModels</code></td>
</tr>
</tbody>
</table>
4.1 Re-use of the single equilibrium calculation state

The Thermo-Calc core keeps an internal state containing the data from previously performed calculations (such as composition of sublattices, previously formed phases, ...). This will be used for start values of future calculations (if not explicitly overwritten) and can strongly influence their convergence and calculation time. It can be useful to save and restore later the core-state in advanced use cases, these include:

- Improving the convergence speed in case of very complicated equilibria if a similar equilibrium had been calculated already before. Similarity refers here primarily to composition, temperature and entered phase set. This case can occur for example with the Nickel-database TCNi.
- Convenient and fast switching between states that have changed a lot (for example regarding suspended phases, numerical settings, ...)

The mechanism of saving and restoring the state is called bookmarking and is controlled with the two methods `tc_python.single_equilibrium.SingleEquilibriumCalculation.bookmark_state()` and `tc_python.single_equilibrium.SingleEquilibriumCalculation.set_state_to_bookmark()`. The following short example demonstrates how to switch between two different states easily in practice:

```python
from tc_python import *

with TCPython() as session:
    calc = (session.
            select_database_and_elements("FEDEMO", ["Fe", "C"]).
            get_system().
            with_single_equilibrium_calculation().
            set_condition(ThermodynamicQuantity.temperature(), 2000.0).
            set_condition("X(C)", 0.01))

    calc.calculate()
    bookmark_temp_condition = calc.bookmark_state()

    calc.set_phase_to_fixed("BCC", 0.5)
    calc.remove_condition(ThermodynamicQuantity.temperature())
    bookmark_fixed_phase_condition = calc.bookmark_state()

    result_temp = calc.set_state_to_bookmark(bookmark_temp_condition)
    print("Conditions do contain temperature: {}\n".format(result_temp.get_conditions()))
    # this calculation had already been performed
    print("Stable phases (do not contain BCC): {}\n".format(result_temp.get_stable_phases()))
```

(continues on next page)
result_fixed_phase = calc.set_state_to_bookmark(bookmark_fixed_phase_condition)
print("Conditions do not contain temperature: {}\).format(result_fixed_phase.get_
→conditions()))
# this calculation had **not yet** been performed
print("Stable phases (do contain BCC): {}\).format(calc.calculate().get_stable_
→phases()))

### 4.2 Re-use and saving of results

Before a calculation is run in TC-Python, a check is made to see if the exact same calculation has run before, and if
that is the case, the result from the calculation can be loaded from disk instead of being re-calculated.

This functionality is always enabled within a script running TC-Python, but you can make it work the same way when
re-running a script, or even when running a completely different script.

To use results from previous calculations, set a folder where TC-Python saves results, and looks for previous results.
This is controlled by the method `tc_python.server.SetUp.set_cache_folder()`.

```python
from tc_python import *
with TCPython() as start:
    start.set_cache_folder("cache")
```

This folder can be a network folder and shared by many users. The calculation is not re-run if there is a previous
TC-Python calculation with the same cache folder and exactly the same settings; the result is instead loaded from disk.

Another possibility is to explicitly save the result to disk and reload it later:

```python
from tc_python import *
with TCPython() as start:
    # ... the system and calculator are set up and the calculation is performed
    result = calculator.calculate()
    result.save_to_disk("./result_dir")
```

You can then load the result again in another session:

```python
from tc_python import *
with TCPython() as start:
    result = SetUp().load_result_from_disk().diffusion("./result_dir")
    x, frac = result.get_mole_fraction_of_component_at_time("Cr", 1000.0)
```
4.3 All TC-Python objects are non-copyable

*Never create a copy* of an instance of a class in TC-Python, neither by using the Python built-in function `deepcopy()` nor in any other way. All classes in TC-Python are proxies for classes in the underlying calculation server and normally hold references to result files. A copied class object in Python would consequently point to the same classes and result files in the calculation server.

Instead of making a copy, always create a new instance:

```python
from tc_python import *

with TCPython() as start:
    system = start.select_database_and_elements("FEDEMO", ["Fe", "Cr"]).get_system()
    calculator = system.with_single_equilibrium_calculation()

    # *do not* copy the `calculator` object, create another one instead
    calculator_2 = system.with_single_equilibrium_calculation()

    # now you can use both calculators for different calculations ...
```

4.4 Python Virtual Environments

A Python installation can have several virtual environments. You can think of a virtual environment as a collection of third party packages that you have access to in your Python scripts. `tc_python` is such a package.

To run TC-Python, you need to *install it into the same virtual environment* as your Python scripts are running in. If your scripts fail on `import tc_python`, you need to execute the following command in the terminal of the same Python environment as your script is running in:

```
pip install TC_Python-<version>-py3-none-any.whl
```

If you use the PyCharm IDE, you should do that within the Terminal built into the IDE. This Terminal runs automatically within your actual (virtual) environment.

To prevent confusion, it is recommended in most cases to *install TC-Python within your global interpreter*, for example by running the `pip install` command within your default Anaconda prompt.

4.5 Using `with TCPython()` efficiently

Normally you should call `with TCPython()` only once within each process.

**Note:** When leaving the `with`-clause, the Java backend engine process is stopped and all temporary data is deleted. Finally when entering the next `with`-clause a new Java process is started. This can take several seconds.

If appropriate, it is safe to run `with TCPython()` in a loop. **Due to the time it takes this only makes sense if the calculation time per iteration is longer than a minute.**

To prevent calling `with TCPython()` multiple times and cleaning up temporary data, you can use the following pattern. **Example:**
from tc_python import *

# ...  

def calculation(calculator):
    # you could also pass the 'session' or 'system' object if more appropriate
    calculator.set_condition("W(Cr)", 0.1)
    # further configuration ...

    result = calculator.calculate()
    # ...
    result.invalidate()  # if the temporary data needs to be cleaned up immediately

if __name__ == '__main__':
    with TCPython() as session:
        system = session.select_database_and_elements("FEDEMO", ["Fe", "Cr"]).get_system()
        calculator = system.with_single_equilibrium_calculation()

        for i in range(50):
            calculation(calculator)

4.6 Parallel calculations

It is possible to perform parallel calculations with TC-Python using multi-processing.

Note: Please note that multi-threading is not suitable for parallelization of computationally intensive tasks in Python. Additionally the Thermo-Calc core is not thread-safe. Using suitable Python-frameworks it is also possible to dispatch the calculations on different computers of a cluster.

A general pattern that can be applied is shown below. This code snippet shows how to perform single equilibrium calculations for different compositions in parallel. In the same way all other calculators of Thermo-Calc can be used or combined. For performance reasons in a real application, probably numpy arrays instead of Python arrays should be used.

Example:

import concurrent.futures

from tc_python import *

def do_perform(parameters):
    # this function runs within an own process
    with TCPython() as start:
        elements = ["Fe", "Cr", "Ni", "C"]
        calculation = (start.select_database_and_elements("FEDEMO", elements).
                        get_system().
                        with_single_equilibrium_calculation().
                        set_condition("T", 1100).
                        set_condition("W(C)", 0.1 / 100).
                        set_condition("W(Ni)", 2.0 / 100))
phase_fractions = []
cr_contents = range(parameters["cr_min"],
        parameters["cr_max"],
        parameters["delta_cr"])
for cr in cr_contents:
    result = (calculation.
        set_condition("W(Cr)", cr / 100).
        calculate())
    phase_fractions.append(result.get_value_of("NPM(BCC_A2)"))
return phase_fractions

if __name__ == "__main__":
    parameters = [
        {"index": 0, "cr_min": 10, "cr_max": 15, "delta_cr": 1},
        {"index": 1, "cr_min": 15, "cr_max": 20, "delta_cr": 1}
    ]
bcc_phase_fraction = []
num_processes = 2

    with concurrent.futures.ProcessPoolExecutor(num_processes) as executor:
        for result_from_process in zip(parameters, executor.map(do_perform,
                        parameters)):
            # params can be used to identify the process and its parameters
            params, phase_fractions_from_process = result_from_process
            bcc_phase_fraction.extend(phase_fractions_from_process)
    # use the result in `bcc_phase_fraction`, for example for plotting

4.7 Handling crashes of the calculation engine

In some cases the Thermo-Calc calculation engine can crash. If batch calculations are performed, this brings down the complete batch. To handle this situation there is an exception you can use.

UnrecoverableCalculationException

That exception is thrown if the calculation server enters a state where no further calculations are possible. You should catch that exception outside of the with TCPython() clause and continue within a new with-clause.

Example:

from tc_python import *
for temperature in range(900, 1100, 10):
    try:
        with TCPython() as start:
            diffusion_result = (start.
                select_thermodynamic_and_kinetic_databases_with_elements("FEDEMO", "MFEDEMO", ["Fe", "Ni"]).
                (continues on next page)
get_system().
with_isothermal_diffusion_calculation().
    set_temperature(temperature).
    set_simulation_time(108000.0).
    add_region(Region("Austenite")
        with_grid(CalculatedGrid.linear().set_no_of_points(50))
        with_composition_profile(CompositionProfile()
            add("Ni", ElementProfile.linear(10.0, 50.0))
        )
    ).
    add_phase("FCC_A1")
)
calculate()

distance, ni_fraction = diffusion_result.get_mass_fraction_of_component_at_time("Ni", 108000.0)
print(ni_fraction)
except UnrecoverableCalculationException as e:
    print('Could not calculate. Continuing with next...')

4.8 Using TC-Python within a Jupyter Notebook or the Python console

TC-Python can also be used from within an interactive Jupyter Notebook and a Python console as well as similar products. The main difference from a regular Python program is that it is not recommended to use a with-clause to manage the TC-Python resources. That is only possible within a single Jupyter Notebook cell. Instead the standalone functions `tc_python.server.start_api_server()` and `tc_python.server.stop_api_server()` should be used for manually managing the resources.

**Note:** The resources of TC-Python are primarily the Java-process running on the backend side that performs the actual calculations and the temporary-directory of TC-Python that can grow to a large size over time, especially if precipitation calculations are performed. If a with-clause is used, these resources are automatically cleared after use.

**You need to make sure that you execute the two functions** `tc_python.server.start_api_server()` and `tc_python.server.stop_api_server()` **exactly once within the Jupyter Notebook session.** If not stopping TC-Python, extra Java-processes might be present and the temporary disk-space is not cleared. However, these issues can be resolved manually.

The temporary directories of TC-Python are named, for example, TC_TMP4747588488953835507 that has a random ID. The temporary directory on different operating systems varies according to the pattern shown in the table.

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Temporary directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>C:\Users{UserName}\AppData\Local\Temp\TC_TMP4747588488953835507</td>
</tr>
<tr>
<td>MacOS</td>
<td>/var/folders/g7/7du01ti_b7mm84n184fn3k9100001g/T/TC_TMP4747588488953835507</td>
</tr>
<tr>
<td>Linux</td>
<td>/tmp/TC_TMP4747588488953835507</td>
</tr>
</tbody>
</table>

In a Jupyter Notebook some features of an IDE such as auto-completion (*TAB-key*), available method lookup (press . and then *TAB*) and parameter lookup (set the cursor within the method-parenthesis and press *SHIFT + TAB* or *SHIFT + TAB + TAB* for the whole docstring) are also available.
Example using TC-Python with a Jupyter Notebook:

```
import * from tc_python
start_api_server()

system = set_up().select_database_and_elements("FREEMAG", ["Fe", "Ni", "Cr"]).set_system()

calc = system.with_single_equilibrium_calculation()

temp = 825 # in K
ni_conc = 10.0 # in wt-%
cr_conc = 8.0 # in wt-%

result = calc.calculate()
```

4.9 Property Model Framework

4.9.1 Debugging Property Model code

You can debug property models while running them from Thermo-Calc.

- Start Thermo-Calc and create a Property Model calculator.

- Select the model you want to debug and check the debug checkbox in the lower right corner of the Python code tab.

```
normalized_driving_force

import _property_model_proxy

model = _property_model_proxy.ModelProxy().java_port=57334, python_port=65247
```

Now the model that you want to debug has been updated with code needed to connect with Thermo-Calc.

- Start debugging the model in the IDE of your choice.

**Note:** You must use a Python interpreter where TC-Python is installed.

In PyCharm it looks like this:
Note: When your IDE and Thermo-Calc have successfully connected, you will see this in the Thermo-Calc log:

```
10:34:42,170 INFO Waiting for developer(!) to start Python process in debugger...
10:34:42,171 INFO Connected successfully to the Python process for the model 'DrivingForcePythonModel' in DEBUG mode
```

You can stop the debug session in your IDE, change the model code, and start debugging again. The changes you made will take effect in Thermo-Calc without the need to restart. If you for instance changed the method `evaluate_model()`, the change will take effect the next time you press `Perform`.

It is also possible to start the models from TC-Python. The workflow is exactly the same as described above, except instead of starting Thermo-Calc graphical user interface, you start a Python script and use the parameter `debug_model=True` when selecting your model.

```python
from tc_python import *

with TCPython() as start:
    property_model = (start.
        select_database_and_elements("FEDEMO", ["Fe", "C"]).
        get_system().
        with_property_model_calculation("my own Driving Force", debug_model=True).
        set_composition("C", 1.0).
    )

    property_model.calculate()

... 
```
4.9.2 Developing Property Models in several files

You can split your Property Model code in several .py files, and there are two ways of doing that:

- **side-by-side** modules
- **common** modules

*Side-by-side* modules are Python files located in the same folder as the Property Model.

*Common* modules are Python files located in a folder outside of the Property Model folder, which makes it possible to share them with several models as a common library.

### 4.9.2.1 side-by-side modules

You are required to:

- Add a `__init__.py` file to your Property Model folder
- Add all imports of *side-by-side* modules in your main Property Model Python file also to the `__init__.py` file

**Example:**

`CriticalTemperaturesPythonModel.py` (The main Property Model file):

```python
from CriticalTemperaturesPython import CriticalTemperatures
from tc_python import *
import numpy as np

class CriticalTemperaturesPythonModel(PropertyModel):
    ...
```

`__init__.py`:

```python
from CriticalTemperaturesPython.critical_temperatures_library import *
from CriticalTemperaturesPython import CriticalTemperatures
```

If you are using PyCharm, the package name of the Property Model might be highlighted as an error, in this case you can mark the Property Model directory (i.e. the root of the present model directory) by right-clicking on it in the project window of PyCharm and marking it as *Sources Root*:

```python
from tc_python import *
import numpy as np
from scipy import optimize
from enum import Enum

class CriticalTemperatures(Enum):
    ...
```

`critical_temperatures_library.py`:
Note: Modules installed in the Python interpreter such as `numpy`, `scipy`, etc can be imported as normal. This only concerns files imported as side-by-side modules.

### 4.9.2.2 common modules

`common` modules work very similar to side-by-side modules except the import statements are done in the “main” `__init__.py` file in `Property Model directory`.

You are required to:

- Add a `__init__.py` file to your property model folder.
- Add all imports of `common` modules in your main property model python file also to both the `__init__.py` file in `Property Model directory` AND the `__init__.py` of the property model.

**Example:**

`CriticalTemperaturesPythonModel.py` (The main Property Model file):

```python
from PropertyModels import Martensite
from tc_python import *

class CriticalTemperaturesPythonModel(PropertyModel):
...
```

`__init__.py`: (The init file located in the property model folder)

```python
from PropertyModels import Martensite
```

`__init__.py`: (The init file located in `Property Model directory`)

```python
from PropertyModels.common.martensite_library import Martensite
```

The file `critical_temperatures_library.py` should in this example be located in a folder called `common` in the `Property Model directory`.

`critical_temperatures_library.py`:

```python
from tc_python import *
import numpy as np
from scipy import optimize
from enum import Enum

class CriticalTemperatures(object):
...
```

**Note:** `common` modules don’t have to be located in folder called `common`. It can be any name, as long as the imports match the folder name.

**Note:** In this example the `Property Model directory` is called ‘PropertyModels’. If you use a different directory for your property model, your imports have to match that.
4.9.3 Alternative Python for Property Models

4.9.3.1 Default bundled Python interpreter

Thermo-Calc is by default using a Python 3.7.2 interpreter bundled to the software for running the property models. It is containing the following major packages:

<table>
<thead>
<tr>
<th>Package</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>matplotlib</td>
<td>3.3.2</td>
</tr>
<tr>
<td>numpy</td>
<td>1.19.2</td>
</tr>
<tr>
<td>scikit-learn</td>
<td>0.23.2</td>
</tr>
<tr>
<td>scipy</td>
<td>1.5.2</td>
</tr>
<tr>
<td>TC-Python</td>
<td>2022b</td>
</tr>
</tbody>
</table>

Warning: Any changes to the interpreter packages can therefore break Thermo-Calc and should be avoided. If the installation gets broken, it can be fixed by reinstalling Thermo-Calc after having removed it.

Please contact the Thermo-Calc support if you think that further packages might be useful in future releases. If these packages are insufficient for you, it is possible to use another Python-interpreter: Configuring another Python interpreter.

The interpreter is located in different places depending on the platform:

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Path to the bundled Python-interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>C:\Program Files\Thermo-Calc\2022b\python\python.exe</td>
</tr>
<tr>
<td>Linux</td>
<td>/home/UserName/Thermo-Calc/2022b/python/bin/python3</td>
</tr>
<tr>
<td>MacOS</td>
<td>/Applications/Thermo-Calc-2022b.app/Contents/Resources/python/bin/python3</td>
</tr>
</tbody>
</table>

4.9.3.2 Configuring another Python interpreter

If you require additional Python-packages or prefer to use your own interpreter installed on your system, you can change the interpreter used by Thermo-Calc to run the property models. Select Tools→Options in the Thermo-Calc GUI and modify the path to that of your Python 3 interpreter of choice:

![Property model Python interpreter: C:\Program Files\Thermo-Calc\2022b\python\python.exe Modify]

4.10 Process Metallurgy Calculations

4.10.1 Equilibrium calculations with changing elements between calculations

It is possible to add, change or remove additions after performing an equilibrium calculation using `tc_python.process_metallurgy.equilibrium.EquilibriumCalculation.calculate()`. This will change the elements being present in the system if the elements of the additions are differing. The Process Metallurgy Module will handle this situation by reloading the database with the latest set of elements. While this is an appropriate approach
in most cases, there can be some disadvantages: reloading the database takes some time and the internal engine state is lost, which may lead to successive calculations failures in some situations.

To avoid the database reload, it is possible to add the respective elements to additions being present in all calculations (with a zero-fraction):

```python
from tc_python import *

with TCPython() as session:
    calc = session.with_metallurgy().with_adiabatic_equilibrium_.
    calculation(ProcessDatabase.OXDEMO)

    # add the element Al with zero-fraction already
    steel = EquilibriumAddition({'Fe': None, 'C': 4, 'Al': 0}, amount=100.0e3,
    temperature=1700 + 273.15)
    slag = EquilibriumAddition({'CaO': 70, 'SiO2': 30}, amount=3.0e3,
    temperature=1700 + 273.15)

    al_addition = EquilibriumAddition({'Al': 100}, amount=1.0e3)

    (calc
        .add_addition(steel)
        .add_addition(slag))

    result_1 = calc.calculate()

    calc.add_addition(al_addition)

    result_2 = calc.calculate()

    # evaluate the result as required ...
```

Or to add a later addition already before the first call to `calculate()` with a zero amount:

```python
from tc_python import *

with TCPython() as session:
    calc = session.with_metallurgy().with_adiabatic_equilibrium_.
    calculation(ProcessDatabase.OXDEMO)

    steel = EquilibriumAddition({'Fe': None, 'C': 4}, amount=100.0e3,
    temperature=1700 + 273.15)
    slag = EquilibriumAddition({'CaO': 70, 'SiO2': 30}, amount=3.0e3,
    temperature=1700 + 273.15)

    al_addition = EquilibriumAddition({'Al': 100}, amount=0)

    (calc
        .add_addition(al_addition)
        .add_addition(steel)
        .add_addition(slag))

    result_1 = calc.calculate()

    calc.update_addition(al_addition.set_amount(1.0e3))

    result_2 = calc.calculate()

    # evaluate the result as required ...
```
4.10.2 Zones

TC-Python is providing a framework for building time-dependent kinetic simulations of industrial and academic metallurgical processes where liquid phases are important. It is based on an Effective Equilibrium Reaction Zone (EERZ) approach which is separating a process into different zones. These zones have identical temperature and composition and are called bulk zones. Such zones can be in contact and react with each other by reaction zones. That means a reaction zone is modelling the interface between two bulk zones. One bulk zone is typically the steel melt and another bulk zone the top slag.

4.10.3 Applications

While this approach can in principle be extended to any number of zones, in the current release TC-Python is providing only one reaction zone. Practical work has however proven that this limitation is not critical for a lot of industrial processes, including ladle furnaces, AOD- and VOD-converters. Even more processes can be modelled with some limit of accuracy.

The reason for the power of the current implementation is that a number of important process features can be included:

- heating (`tc_python.process_metallurgy.process.Zone.add_power()`)
- cooling (`tc_python.process_metallurgy.process.Zone.add_power()`)
- heat transfer between bulk zones (`tc_python.process_metallurgy.process.ReactionZone.add_heat_transfer()`)  
- inclusion formation
- inclusion flotation and other transfer of phase groups between bulk zones (`tc_python.process_metallurgy.process.ReactionZone.add_transfer_of_phase_group()`)  
- addition of material and gas at any time in any zone (`tc_python.process_metallurgy.process.Zone.add_addition()`)  
- an exhaust gas zone collecting all formed gas (`tc_python.process_metallurgy.process.ProcessSimulationResult.get_exhaust_gas()`)  
- time-dependent definition of most parameters (e.g., mass transfer coefficient, transfer of phase group, heating, etc.)

Please note that many of these features are called as well a reaction zone in other EERZ model implementations.

4.10.4 Implementation of practical process models

The Process Metallurgy Module has been successfully applied to a number of industrial processes.

Due to the broad range of industrial metallurgical processes, TC-Python is not providing ready-to-use models for certain processes. There are however examples available for common processes and this collection will be extended over time. The implementation of a model is an abstraction of the real process and should always be kept as simple as possible. Practical experience has proven that in many situations not more than one reaction zone is required.

The mass transfer coefficient is a fundamental parameter describing the kinetics in a reaction zone and is generally an empirical parameter. It depends however mostly on the geometry and stirring conditions in the process and not on the material compositions. Further on, the mass transfer coefficient has usually typical values for a given process - regardless of the actual furnace. That means that existing suggestions from the literature can be used as a starting point to derive the actual mass transfer coefficient for the process of interest.
5.1 Calculations

5.1.1 Module “single_equilibrium”

```python
class tc_python.single_equilibrium.AbstractSingleEquilibriumCalculation(calculator):
    Bases: tc_python.abstract_base.AbstractCalculation

    Abstract configuration required for a single equilibrium calculation.

    Note: This is an abstract class that cannot be used directly.
```

disable_global_minimization()
    Turns the global minimization completely off.

    Returns This SingleEquilibriumCalculation object

enable_global_minimization()
    Turns the global minimization on (using the default settings).

    Returns This SingleEquilibriumCalculation object

get_components() \(\rightarrow\) List[str]
    Returns a list of components in the system (including all components auto-selected by the database(s)).

    Returns The components

get_gibbs_energy_addition_for(phase: str) \(\rightarrow\) float
    Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

    It is not composition-, temperature- or pressure-dependent.

    Parameters phase – Specify the name of the (stoichiometric or solution) phase with the addition

    Returns Gibbs energy addition to G per mole formula unit.

get_system_data() \(\rightarrow\) tc_python.abstract_base.SystemData
    Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

    Note: Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.
Returns The system data

```python
run_poly_command(command: str)
```
Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Parameters `command` – The Thermo-Calc Console Mode command

Returns This `SingleEquilibriumCalculation` object

**Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

**Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

```python
set_component_to_entered(component: str)
```
Sets the specified component to the status ENTERED, that is the default state.

Parameters `component` – The component name or `ALL_COMPONENTS`

Returns This `SingleEquilibriumCalculation` object

```python
set_component_to_suspended(component: str, reset_conditions: bool = False)
```
Sets the specified component to the status SUSPENDED, i.e. it is ignored in the calculation.

Parameters
- `reset_conditions` – if ‘True’ also remove composition conditions for the component if they are defined
- `component` – The component name or `ALL_COMPONENTS`

Returns This `SingleEquilibriumCalculation` object

```python
set_gibbs_energy_addition_for(phase: str, gibbs_energy: float)
```
Used to specify the additional energy term (always being a constant) of a given phase. The value (gibbs_energy) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

Parameters
- `phase` – Specify the name of the (stoichiometric or solution) phase with the addition
- `gibbs_energy` – Addition to G per mole formula unit

Returns This `SingleEquilibriumCalculation` object

```python
set_phase_to_dormant(phase: str)
```
Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

Parameters `phase` – The phase name or `ALL_PHASES` for all phases

Returns This `SingleEquilibriumCalculation` object

```python
set_phase_to_entered(phase: str, amount: float = 1.0)
```
Sets the phase to the status ENTERED, that is the default state.

Parameters
set_phase_to_fixed(phase: str, amount: float)
Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

Parameters

- **phase** – The phase name
- **amount** – The fixed phase fraction (between 0.0 and 1.0)

Returns This `SingleEquilibriumCalculation` object

set_phase_to_suspended(phase: str)
Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

Parameters phase – The phase name or `ALL_PHASES` for all phases

Returns This `SingleEquilibriumCalculation` object

with_options(options: tc_python.single_equilibrium.SingleEquilibriumOptions)
Sets the simulation options.

Parameters options – The simulation options

Returns This `SingleEquilibriumCalculation` object

with_reference_state(component: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0)
The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

Parameters

- **component** – The name of the element must be given.
- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
• pressure – The Pressure (in Pa) for the reference state.

Returns This SingleEquilibriumCalculation object

with_system_modifications (system_modifications: tc_python.abstract_base.SystemModifications)
Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a *.tdb-file.

Parameters system_modifications – The system modification to be performed

Returns This SingleEquilibriumCalculation object

class tc_python.single_equilibrium.SingleEquilibriumCalculation (calculator)
Bases: tc_python.single_equilibrium.AbstractSingleEquilibriumCalculation
Configuration for a single equilibrium calculation.

Note: Specify the conditions and possibly other settings, the calculation is performed with calculate().

bookmark_state (bookmark_id: str = '') → str
Puts a “bookmark” on the current calculation-state of the calculator allowing the program to return to this state later as needed.

By bookmarking a state, you can simplify the convergence of equilibria when they strongly depend on the starting conditions (i.e. the state). Also use it to improve performance by running a calculation, then bookmarking it, and later returning to it for other equilibria whose conditions are “close” to the bookmarked equilibrium.

This method is used in combination with the method set_state_to_bookmark().

Parameters bookmark_id – The bookmark id. If omitted a generated id is used and returned

Returns The bookmark id

calculate () → tc_python.single_equilibrium.SingleEquilibriumTempResult
Performs the calculation and provides a temporary result object that is only valid until something gets changed in the calculation state. The method calculate() is the default approach and should be used in most cases.

Returns A new SingleEquilibriumTempResult object which can be used to get specific values from the calculated result. It is undefined behavior to use that object after the state of the calculation has been changed.

Warning: If the result object should be valid for the whole program lifetime, use calculate_with_state() instead.

calculate_with_state (timeout_in_minutes: float = 0.0) → tc_python.single_equilibrium.SingleEquilibriumResult
Performs the calculation and provides a result object that reflects the present state of the calculation during the whole lifetime of the object.
Note: Because this method has performance and temporary disk space overhead (i.e. it is resource heavy), only use it when it is necessary to access the result object after the state is changed. In most cases you should use the method \texttt{calculate()}.

\textbf{Parameters} \texttt{timeout\_in\_minutes} – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than \texttt{timeout\_in\_minutes}, a \texttt{UnrecoverableCalculationException} will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

\textbf{Returns} A new \texttt{SingleEquilibriumResult} object which can be used later at any time to get specific values from the calculated result.

disable\_global\_minimization()

Turns the global minimization completely off.

\textbf{Returns} This \texttt{SingleEquilibriumCalculation} object

enable\_global\_minimization()

Turns the global minimization on (using the default settings).

\textbf{Returns} This \texttt{SingleEquilibriumCalculation} object

\textbf{get\_components()} → List[str]

Returns a list of components in the system (including all components auto-selected by the database(s)).

\textbf{Returns} The components

gibb\_energy\_add\_for(phase: str) → float

Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

\textbf{Parameters} \texttt{phase} – Specify the name of the (stoichiometric or solution) phase with the addition

\textbf{Returns} Gibbs energy addition to G per mole formula unit.

gibb\_energy\_add\_for(phase: str) → float

\textbf{get\_interfacial\_energy}(matrix\_phase: str, precipitate\_phases: List[str], zero\_volume\_elements: List[str] = ['C', 'N']) → Dict[str, float]

Estimates the interfacial energy between a matrix phase and a precipitate phase using thermodynamic data from a CALPHAD database. The approximation model is based on Becker’s bond energy approach.

\textbf{Default:} elements with no contribution to volume are C and N.

\textbf{Parameters}

- \texttt{matrix\_phase} – The matrix phase.

- \texttt{precipitate\_phases} – The list of precipitate phases for which interfacial energy between them and the matrix phase is to be calculated.

- \texttt{zero\_volume\_elements} – The elements that are assumed to not contribute to the volume.

\textbf{Returns} A dictionary containing interfacial energy per precipitate phase.

gibb\_energy\_add\_for(phase: str) → float

\textbf{get\_system\_data()} → \texttt{tc\_python.abstract\_base.SystemData}

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using \texttt{with\_system\_modifications()}.  

\section*{5.1. Calculations}
Note: Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.

Returns The system data

remove_all_conditions()  
Removes all set conditions.

Returns This SingleEquilibriumCalculation object

remove_condition(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str])  
Removes the specified condition.

Parameters quantity – the thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example “X(Cr)”)  

Returns This SingleEquilibriumCalculation object

run_poly_command(command: str)  
Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Parameters command – The Thermo-Calc Console Mode command  

Returns This SingleEquilibriumCalculation object

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

set_component_to_entered(component: str)  
Sets the specified component to the status ENTERED, that is the default state.

Parameters component – The component name or ALL_COMPONENTS  

Returns This SingleEquilibriumCalculation object

set_component_to_suspended(component: str, reset_conditions: bool = False)  
Sets the specified component to the status SUSPENDED, i.e. it is ignored in the calculation.

Parameters

- reset_conditions – if ‘True’ also remove composition conditions for the component if they are defined  
- component – The component name or ALL_COMPONENTS

Returns This SingleEquilibriumCalculation object

set_condition(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], value: float)  
Sets the specified condition.

Parameters

- quantity – The thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example “X(Cr)”)

Chapter 5. API Reference
• **value** – The value of the condition

**Returns** This *SingleEquilibriumCalculation* object

**set_gibbs_energy_addition_for**(phase: str, gibbs_energy: float)

Used to specify the additional energy term (always being a constant) of a given phase. The value *(gibbs_energy)* given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters**

• **phase** – Specify the name of the (stoichiometric or solution) phase with the addition

• **gibbs_energy** – Addition to G per mole formula unit

**Returns** This *SingleEquilibriumCalculation* object

**set_phase_to_dormant**(phase: str)

Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

**Parameters** phase – The phase name or *ALL_PHASES* for all phases

**Returns** This *SingleEquilibriumCalculation* object

**set_phase_to_entered**(phase: str, amount: float = 1.0)

Sets the phase to the status ENTERED, that is the default state.

**Parameters**

• **phase** – The phase name or *ALL_PHASES* for all phases

• **amount** – The phase fraction (between 0.0 and 1.0)

**Returns** This *SingleEquilibriumCalculation* object

**set_phase_to_fixed**(phase: str, amount: float)

Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

**Parameters**

• **phase** – The phase name

• **amount** – The fixed phase fraction (between 0.0 and 1.0)

**Returns** This *SingleEquilibriumCalculation* object

**set_phase_to_suspended**(phase: str)

Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

**Parameters** phase – The phase name or *ALL_PHASES* for all phases

**Returns** This *SingleEquilibriumCalculation* object

**set_state_to_bookmark**(bookmark_id: str) → tc_python.single_equilibrium.SingleEquilibriumTempResult

Resets the calculation state to a previously bookmarked state.

After calling this method, the calculation behaves exactly as it would after the bookmarked calculation ran.

This method is used in combination with the method *bookmark_state()*.

**Parameters** bookmark_id – The bookmark id of the state to return to.
Returns A new `SingleEquilibriumTempResult` object which can be used to get specific values from the calculated result. It is undefined behavior to use that object after the state of the calculation has changed.

`with_options(options: tc_python.single_equilibrium.SingleEquilibriumOptions)`
Sets the simulation options.

**Parameters**
- `options` – The simulation options

**Returns** This `SingleEquilibriumCalculation` object

`with_reference_state(component: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0)`
The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

**Parameters**
- `component` – The name of the element must be given.
- `phase` – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
- `temperature` – The Temperature (in K) for the reference state. Or `CURRENT_TEMPERATURE` which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
- `pressure` – The Pressure (in Pa) for the reference state.

**Returns** This `SingleEquilibriumCalculation` object

`with_system_modifications(system_modifications: tc_python.abstract_base.SystemModifications)`
Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

**Note:** This is only possible if the system has been read from unencrypted (i.e. `user`) databases loaded as a `.tdb`-file.

**Parameters**
- `system_modifications` – The system modification to be performed

**Returns** This `SingleEquilibriumCalculation` object
class tc_python.single_equilibrium.SingleEquilibriumOptions

General simulation conditions for the thermodynamic calculations.

disable_approximate_driving_force_for_metastable_phases()
Disables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.
If it is important that these driving forces are correct, use disable_approximate_driving_force_for_metastable_phases() to force the calculation to converge for the metastable phases.

Returns This SingleEquilibriumOptions object

disable_control_step_size_during_minimization()
Disables stepsize control during minimization (non-global).

Default: Enabled

Returns This SingleEquilibriumOptions object

disable_force_positive_definite_phase_hessian()
Disables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for “Hessian minimization”.

Default: Enabled

Returns This SingleEquilibriumOptions object

enable_approximate_driving_force_for_metastable_phases()
Enables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.
If it is important that these driving forces are correct, use disable_approximate_driving_force_for_metastable_phases() to force the calculation to converge for the metastable phases.

Returns This SingleEquilibriumOptions object

enable_control_step_size_during_minimization()
Enables stepsize control during normal minimization (non-global).

Default: Enabled

Returns This SingleEquilibriumOptions object
enable_force_positive_definite_phase_hessian()

Enables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for “Hessian minimization”.

Default: Enabled

Returns This `SingleEquilibriumOptions` object

set_global_minimization_max_grid_points (max_grid_points: int = 2000)

Sets the maximum number of grid points in global minimization. **Only applicable if global minimization is actually used.**

Default: 2000 points

Parameters max_grid_points – The maximum number of grid points

Returns This `SingleEquilibriumOptions` object

set_max_no_of_iterations (max_no_of_iterations: int = 500)

Set the maximum number of iterations.

Default: max. 500 iterations

Note: As some models give computation times of more than 1 CPU second/iteration, this number is also used to check the CPU time and the calculation stops if 500 CPU seconds/iterations are used.

Parameters max_no_of_iterations – The max. number of iterations

Returns This `SingleEquilibriumOptions` object

set_required_accuracy (accuracy: float = 1e-06)

Sets the required relative accuracy.

Default: 1.0E-6

Note: This is a relative accuracy, and the program requires that the relative difference in each variable must be lower than this value before it has converged. A larger value normally means fewer iterations but less accurate solutions. The value should be at least one order of magnitude larger than the machine precision.

Parameters accuracy – The required relative accuracy

Returns This `SingleEquilibriumOptions` object

set_smallest_fraction (smallest_fraction: float = 1e-12)

Sets the smallest fraction for constituents that are unstable.

It is normally only in the gas phase that you can find such low fractions.

The default value for the smallest site-fractions is 1E-12 for all phases except for IDEAL phase with one sublattice site (such as the GAS mixture phase in many databases) for which the default value is always as 1E-30.

Parameters smallest_fraction – The smallest fraction for constituents that are unstable

Returns This `SingleEquilibriumOptions` object
class tc_python.single_equilibrium.SingleEquilibriumResult(result)
Bases: tc_python.abstract_base.AbstractResult

Result of a single equilibrium calculation, it can be evaluated using a Quantity or Console Mode syntax.

change_pressure(pressure: float)
Change the pressure and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with higher performance. The properties are calculated at the new pressure using the phase amount, temperature and composition of phases from the initial equilibrium. Use get_value_of() to obtain them.

Parameters pressure – The pressure [Pa]

Returns This SingleEquilibriumCalculation object

change_temperature(temperature: float)
Change the temperature and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with high performance. The properties are calculated at the new temperature using the phase amount, pressure and composition of phases from the initial equilibrium. Use get_value_of() to obtain them.

Note: This is typically used when calculating room temperature properties (e.g. density) for a material when it is assumed that the equilibrium phase amount and composition freeze-in at a higher temperature during cooling.

Parameters temperature – The temperature [K]

Returns This SingleEquilibriumCalculation object

get_components() → List[str]
Returns the names of the components selected in the system (including any components auto-selected by the database(s)).

Returns The names of the selected components

get_conditions() → List[str]
Returns the conditions.

Returns The selected conditions

get_phases() → List[str]
Returns the phases present in the system due to its configuration. It also contains all phases that have been automatically added during the calculation, this is the difference to the method System.get_phases_in_system().

Returns The names of the phases in the system including automatically added phases

get_stable_phases() → List[str]
Returns the stable phases (i.e. the phases present in the current equilibrium).

Returns The names of the stable phases

get_value_of(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str]) → float
Returns a value from a single equilibrium calculation.

Parameters quantity – The thermodynamic quantity to get the value of; a Console Mode syntax strings can be used as an alternative (for example “NPM(FCC_A1)"

Returns The requested value
run_poly_command(command: str)
Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine. This affects only the state of the result object.

Parameters command – The Thermo-Calc Console Mode command

Returns This SingleEquilibriumCalculation object

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

disable_poly_mode()
Disables the POLY module in the engine.

Parameters:

Note: It is undefined behavior to disable the POLY module after it has been enabled.

save_to_disk(path: str)
Saves the result to disk. Note that the result is a folder, containing potentially many files. The result can later be loaded with load_result_from_disk()

Parameters path – the path to the folder you want the result to be saved in. It can be relative or absolute.

Returns this SingleEquilibriumResult object

class tc_python.single_equilibrium.SingleEquilibriumTempResult(result)
Bases: tc_python.abstract_base.AbstractResult
Result of a single equilibrium calculation that is only valid until something gets changed in the calculation state. It can be evaluated using a Quantity or Console Mode syntax.

Warning: Note that it is undefined behavior to use that object after something has been changed in the state of the calculation, this will result in an InvalidResultStateException exception being raised.

change_pressure(pressure: float)
Change the pressure and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with higher performance. The properties are calculated at the new pressure using the phase amount, temperature and composition of phases from the initial equilibrium. Use get_value_of() to obtain them.

Parameters pressure – The pressure [Pa]

Returns This SingleEquilibriumCalculation object

change_temperature(temperature: float)
Change the temperature and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with high performance. The properties are calculated at the new temperature using the phase amount, pressure and composition of phases from the initial equilibrium. Use get_value_of() to obtain them.

Note: This is typically used when calculating room temperature properties (e.g. density) for a material when it is assumed that the equilibrium phase amount and composition freeze-in at a higher temperature during cooling.

Parameters temperature – The temperature [K]
Returns This SingleEquilibriumCalculation object

get_components() \rightarrow \text{List[str]}
Returns the names of the components selected in the system (including any components auto-selected by the database(s)).

Returns The names of the selected components

Raises InvalidResultStateException – If something has been changed in the state of the calculation since that result object has been created

get_conditions() \rightarrow \text{List[str]}
Returns the conditions.

Returns List containing the selected conditions

Raises InvalidResultStateException – If something has been changed in the state of the calculation since that result object has been created

get_phases() \rightarrow \text{List[str]}
Returns the phases present in the system due to its configuration. It also contains all phases that have been automatically added during the calculation, this is the difference to the method System.get_phases_in_system().

Returns The names of the phases in the system including automatically added phases

Raises InvalidResultStateException – If something has been changed in the state of the calculation since that result object has been created

get_stable_phases() \rightarrow \text{List[str]}
Returns the stable phases (i.e. the phases present in the current equilibrium).

Returns The names of the stable phases

Raises InvalidResultStateException – If something has been changed in the state of the calculation since that result object has been created

get_value_of(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str]) \rightarrow \text{float}
Returns a value from a single equilibrium calculation.

Parameters quantity – The thermodynamic quantity to get the value of; a Console Mode syntax strings can be used as an alternative (for example “NPM(FCC_A1)”)

Returns The requested value

Raises InvalidResultStateException – If something has been changed in the state of the calculation since that result object has been created

run_poly_command(command: str)
Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Parameters command – The Thermo-Calc Console Mode command

Returns This SingleEquilibriumCalculation object

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).
5.1.2 Module “batch_equilibrium”

class tc_python.batch_equilibrium.BatchEquilibriumCalculation(calculator)
Bases: tc_python.abstract_base.AbstractCalculation

Configuration for a series of single equilibrium calculations performed in a vectorized fashion.

Note: Specify the conditions and call calculate().

Tip: The performance of batch equilibrium calculations can be significantly better than looping and using SingleEquilibriumCalculation if the actual Thermo-Calc calculation is fast. There is little advantage if the Thermo-Calc equilibrium calculations take a long time (typically for large systems and databases).

calculate (quantities: List[Union[tc_python.quantity_factory.ThermodynamicQuantity, str]], logging_frequency: int = 10, timeout_in_minutes: float = 0.0) → tc_python.batch_equilibrium.BatchEquilibriumResult

Runs the batch equilibrium calculation. The calculated BatchEquilibriumResult can then be queried for the values of the quantities specified.

Example:

```python
>>> quantities = ['G', 'X(BCC)']
```

Parameters

- quantities – A list of the quantities to be calculated.
- logging_frequency – Determines how often logging should be done.
- timeout_in_minutes – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

Returns A BatchEquilibriumResult which later can be used to get specific values from the calculated result.

disable_global_minimization ()

Turns the global minimization completely off.

Returns This BatchEquilibriumCalculation object

enable_global_minimization()  

Turns the global minimization on (using the default settings).

Returns This BatchEquilibriumCalculation object

get_components() → List[str]

Returns a list of components in the system (including all components auto-selected by the database(s)).

Returns The components

get_gibbs_energy_addition_for (phase: str) → float

Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.
**Parameters** phase – Specify the name of the (stoichiometric or solution) phase with the addition

**Returns** Gibbs energy addition to G per mole formula unit.

**get_system_data()** → `tc_python.abstract_base.SystemData`

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

**Note:** Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb*-file.

**Returns** The system data

**remove_all_conditions()**

Removes all set conditions.

**Returns** This `BatchEquilibriumCalculation` object

**remove_condition(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str])**

Removes the specified condition.

**Parameters** quantity – the thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example “X(Cr)”)

**Returns** This `BatchEquilibriumCalculation` object

**run_poly_command(command: str)**

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

**Parameters** command – The Thermo-Calc Console Mode command

**Returns** This `BatchEquilibriumCalculation` object

**Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

**Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

**set_component_to_entered(component: str)**

Sets the specified component to the status ENTERED, that is the default state.

**Parameters** component – The component name or `ALL_COMPONENTS`

**Returns** This `BatchEquilibriumCalculation` object

**set_component_to_suspended(component: str, reset_conditions: bool = False)**

Sets the specified component to the status SUSPENDED, i.e. it is ignored in the calculation.

**Parameters**

- reset_conditions – if ‘True’ also remove composition conditions for the component if they are defined

- component – The component name or `ALL_COMPONENTS`

**Returns** This `BatchEquilibriumCalculation` object
**set_condition** *(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], value: float)*

Sets the specified condition.

**Parameters**

- **quantity** – The thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example “X(Cr)”)
- **value** – The value of the condition

**Returns** This `BatchEquilibriumCalculation` object

**set_conditions_for_equilibria** *(equilibria: List[List[Tuple[Union[tc_python.quantity_factory.ThermodynamicQuantity, str], float]]])*议案

Set the conditions of the equilibria to be calculated.

This is done by sending a list of equilibria at once.

Each equilibrium itself is a list of conditions that will be changed for that equilibrium.

A condition is described by a tuple containing:

1. A Console Mode syntax string or a `ThermodynamicQuantity` instance,
2. A float value specifying the value of the condition.

**Example:**
```
>>> [[('T', 800), ('X(Cr)', 0.1)], [('T', 850), ('X(Cr)', 0.11)]
```

You can use `ThermodynamicQuantity` instead of a Console Mode syntax string when specifying type of condition.

**Example:**
```
>>> [[(ThermodynamicQuantity.temperature(), 800), (ThermodynamicQuantity.mole_fraction_of_a_component('Cr'), 0.1)], [(ThermodynamicQuantity.temperature(), 850), (ThermodynamicQuantity.mole_fraction_of_a_component('Cr'), 0.15)]
```

**Parameters** **equilibria** – The list of equilibria

**Returns** This `BatchEquilibriumCalculation` object

**set_gibbs_energy_addition_for** *(phase: str, gibbs_energy: float)*

Used to specify the additional energy term (always being a constant) of a given phase. The value *(gibbs_energy)* given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters**

- **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
- **gibbs_energy** – Addition to G per mole formula unit

**Returns** This `BatchEquilibriumCalculation` object

**set_phase_to_dormant** *(phase: str)*

Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

**Parameters** **phase** – The phase name or `ALL_PHASES` for all phases
set_phase_to_entered (phase: str, amount: float = 1.0)
Sets the phase to the status ENTERED, that is the default state.

Parameters
• phase – The phase name or ALL_PHASES for all phases
• amount – The phase fraction (between 0.0 and 1.0)

Returns This BatchEquilibriumCalculation object

set_phase_to_fixed (phase: str, amount: float)
Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

Parameters
• phase – The phase name
• amount – The fixed phase fraction (between 0.0 and 1.0)

Returns This BatchEquilibriumCalculation object

set_phase_to_suspended (phase: str)
Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

Parameters phase – The phase name or ALL_PHASES for all phases

Returns This BatchEquilibriumCalculation object

with_options (options: tc_python.single_equilibrium.SingleEquilibriumOptions)
Sets the simulation options.

Parameters options – The simulation options

Returns This BatchEquilibriumCalculation object

with_reference_state (component: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0)
The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

Parameters
• component – The name of the element must be given.
• **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
• **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
• **pressure** – The Pressure (in Pa) for the reference state.

**Returns** This BatchEquilibriumCalculation object

**with_system_modifications** *(system_modifications: tc_python.abstract_base.SystemModifications)*

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

**Note:** This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a *.tdb*-file.

**Parameters** system_modifications – The system modification to be performed

**Returns** This BatchEquilibriumCalculation object

**class** tc_python.batch_equilibrium.BatchEquilibriumResult *(result)*

**Bases:** object

Result of a batch equilibrium calculation. This can be used to query for specific values.

**get_values_of** *(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str]) ➔ List[float]*

Returns values from a batch equilibrium calculation.

**Warning:** The quantity must be one of the quantities specified for the BatchEquilibriumCalculation object that created the result object.

**Example:**

```python
>>> batch_result = batch_calculation.calculate(quantities = ['G', 'X(BCC)'])
>>> batch_result.get_values_of('G')
```

**Parameters** quantity – the thermodynamic quantity to get the value of; a Console Mode syntax strings can be used as an alternative (for example “NPM(FCC_A1)”)

**invalidate** *

Invalidates the object and frees the disk space used by it.

**Note:** This is only required if the disk space occupied by the object needs to be released during the calculation. No data can be retrieved from the object afterwards.
5.1.3 Module “precipitation”

class tc_python.precipitation.FixedGrainSize(grain_radius: float = 0.0001)
Bases: tc_python.precipitation.GrainGrowthModel

set_grain_aspect_ratio (grain_aspect_ratio: float = 1.0)
Enter a numerical value. Default: 1.0.

Parameters grain_aspect_ratio – The grain aspect ratio [-]

class tc_python.precipitation.GrainGrowth(grain_size_distribution):
Bases: tc_python.precipitation.GrainGrowthModel

disable_zener_pinning ()
Disable Zener pinning to ignore the particle pinning effect on the grain growth. Zener pinning is by default disabled when no grain size distribution is defined, i.e. a single constant grain size is used. The setting is by default enabled when a grain size distribution is defined.

Returns This GrainSizeDistribution object

enable_zener_pinning ()
Enable Zener pinning to simulate the particle pinning effect on the grain growth. The setting is by default enabled when a grain size distribution is defined.

Returns This GrainSizeDistribution object

set_grain_boundary_energy (energy: float = 0.5)
Set the energy of the grain bounday.

Parameters energy – The grain boundary energy [J/m2]

Returns This GrainSizeDistribution object

set_grain_boundary_mobility_activation_energy (activation_energy: float = 242000.0)
Set the grain boundary mobility activation energy where the mobility is defined by an Arrhenius type of equation.

Parameters activation_energy – The mobility activation energy [J/mol]

Returns This GrainSizeDistribution object

set_grain_boundary_mobility_pre_factor (pre_factor: float = 0.004)
Set the grain boundary mobility prefactor where the mobility is defined by an Arrhenius type of equation.

Parameters pre_factor – The grain boundary mobility pre factor [m^4/(J s)]

Returns This GrainSizeDistribution object

class tc_python.precipitation.GrainGrowthModel
Bases: object

Factory class providing objects representing a grain growth model.

classmethod fixed_grain_size (grain_radius: float = 0.0001)
Fixed grain radius size. Default: 1.0E-4 m

Parameters grain_radius – The grain radius / size [m]

classmethod grain_growth (grain_size_distribution: tc_python.precipitation.GrainSizeDistribution)
Sets the initial grain size distribution for the matrix. Default: If the initial grain size distribution is not explicitly provided, a constant average grains size will be used and no grain growth evaluated during the simulation.
Tip: Use this option if you want to study the further evolution of an existing microstructure.

Parameters `grain_size_distribution` – grain size distribution

class tc_python.precipitation.GrainsizeDistribution
    Bases: object

Receps the grain size distribution at a certain time.

`add_radius_and_number_density` (radius: float, number_density: float)

Adds a radius and number density pair to the grain size distribution.

Parameters

- `radius` – The radius [m]
- `number_density` – The number of grains per unit volume per unit length [m^-4]

Returns

This `GrainSizeDistribution` object

class tc_python.precipitation.GrowthRateModel(value)
    Bases: enum.Enum

Choice of the used growth rate model for a precipitate.

The most efficient model is the `Simplified model`, which is the default and applicable to most alloy systems under the assumption that either the supersaturation is small, or the alloying elements have comparable diffusivity. If all alloying elements are substitutional but they have remarkable diffusivity difference, e.g. in Al-Zr system, or if the diffusivity is strongly composition-dependent, the `General model` is preferred. If the supersaturation is high, and meanwhile there are fast-diffusing interstitial elements such as C, the `Advanced model` is more appropriate to capture the NPLE mechanism.

**ADVANCED** = 3

The advanced model has been proposed by Chen, Jeppsson, and Ågren (CJA) (2008) and calculates the velocity of a moving phase interface in multicomponent systems by identifying the operating tie-line from the solution of the flux-balance equations. This model can treat both high supersaturation and cross-diffusion rigorously. Spontaneous transitions between different modes (LE and NPLE) of phase transformation can be captured without any ad-hoc treatment.

**GENERAL** = 5

The general model is based on the Morral-Purdy model, which follows the same quasi-steady state approximation as the Simplified model, but improves it by taking the cross-diffusion into account.

**NPLE** = 11

The Non-Partitioning Local Equilibrium (NPLE) growth rate model is only available for alloy systems where Fe is the major element and at least one interstitial element partitions into the precipitate phase. This model is specifically designed to deal with the fast diffusion of interstitial elements (C, N, etc.) in Fe alloys. Based on the Simplified growth model, it still holds a local equilibrium condition at the migrating interface. It chooses a tie-line under NPLE condition so that the u-fractions of all substitutional elements and minor interstitial elements in the precipitate phase are the same as those in the far-field matrix phase (i.e. the overall instantaneous matrix composition).

**PARA_EQ** = 10

The para-equilibrium model is only available for alloy systems where Fe is the major element and C is
the only interstitial element, which also partitions into the precipitate phase. The interstitial elements, e.g. C, N, etc., usually have remarkably faster diffusion rate than the substitutional elements. Meanwhile, they are assumed to have negligible volume contribution, and as a result the composition variables are replaced by u-fractions when interstitial elements are included in the system. This model is specifically designed to address the fast diffusion of C in Fe alloys. Based on the Simplified growth rate model it holds a para-equilibrium condition at the migrating interface. Contrary to the regular ortho-equilibrium condition state that assumes that all alloying elements are in equilibrium at the interface, the para-equilibrium assumes only equilibrium for C. The substitutional elements are immobile and thus have the same compositions (u-fractions) across the interface.

SIMPLIFIED = 2

The simplified model is based on the advanced model but avoids the difficulty of finding the operating tie-line and uses instead the tie-line across the bulk composition. This is the default growth rate model.

class tc_python.precipitation.MatrixPhase(matrix_phase_name: str)
Bases: object
The matrix phase in a precipitation calculation

add_precipitate_phase (precipitate_phase: tc_python.precipitation.PrecipitatePhase)
Adds a precipitate phase.

Parameters precipitate_phase – The precipitate phase

set_dislocation_density (dislocation_density: float = 5.0E12 m^-2)
Enter a numerical value. Default: 5.0E12 m^-2.

Parameters dislocation_density – The dislocation density [m^-2]

set_grain_aspect_ratio (grain_aspect_ratio: float = 1.0)
Enter a numerical value. Default: 1.0.

Note: Deprecated in version 2022a: Use with_grain_growth_model() instead. This method will be removed in release 2023a.

Parameters grain_aspect_ratio – The grain aspect ratio [-]

set_grain_radius (grain_radius: float = 0.0001)
Sets grain radius / size. Default: 1.0E-4 m

Note: Deprecated in version 2022a: Use with_grain_growth_model() instead. This method will be removed in release 2023a.

Parameters grain_radius – The grain radius / size [m]

set_mobility_enhancement_activation_energy (mobility_enhancement_activation_energy: float = 0.0)
A value that adds to the activation energy of mobility data from the database. Default: 0.0 J/mol

Parameters mobility_enhancement_activation_energy – The value that adds to the activation energy of mobility data from the database [J/mol].

set_mobility_enhancement_prefactor (mobility_enhancement_prefactor: float = 1.0)
A parameter that multiplies to the mobility data from database. Default: 1.0

Parameters mobility_enhancement_prefactor – The mobility enhancement factor [-]
set_molar_volume(volume: float)
Sets the molar volume of the phase.

Default: If not set, the molar volume is taken from the thermodynamic database (or set to 7.0e-6 m^3/mol if the database contains no molar volume information).

Parameters

- volume – The molar volume [m^3/mol]

with_elastic_properties_cubic(c11: float, c12: float, c44: float)
Sets the elastic properties to "cubic" and specifies the elastic stiffness tensor components. Default: if not chosen, the default is DISREGARD

Parameters

- c11 – The stiffness tensor component c11 [GPa]
- c12 – The stiffness tensor component c12 [GPa]
- c44 – The stiffness tensor component c44 [GPa]

with_elastic_properties_disregard()
Set to disregard to ignore the elastic properties. Default: This is the default option

with_elastic_properties_isotropic(shear_modulus: float, poisson_ratio: float)
Sets elastic properties to isotropic. Default: if not chosen, the default is DISREGARD

Parameters

- shear_modulus – The shear modulus [GPa]
- poisson_ratio – The Poisson’s ratio [-]

with_grain_growth_model(grain_growth_model: tc_python.precipitation.GrainGrowthModel)
Sets the model for grain growth. Either fixed size or with a starting distribution

Default: Fixed grain radius size 1.0E-4 m

Parameters

- grain_growth_model – the grain growth model

class tc_python.precipitation.NumericalParameters
Bases: object
Numerical parameters

set_max_overall_volume_change(max_overall_volume_change: float = 0.001)
This defines the maximum absolute (not ratio) change of the volume fraction allowed during one time step. Default: 0.001

Parameters

- max_overall_volume_change – The maximum absolute (not ratio) change of the volume fraction allowed during one time step [-]

set_max_radius_points_per_magnitude(max_radius_points_per_magnitude: float = 200.0)
Sets the maximum number of grid points over one order of magnitude in radius. Default: 200.0

Parameters

- max_radius_points_per_magnitude – The maximum number of grid points over one order of magnitude in radius [-]

set_max_rel_change_critical_radius(max_rel_change_critical_radius: float = 0.1)
Used to place a constraint on how fast the critical radium can vary, and thus put a limit on time step. Default: 0.1

Parameters

- max_rel_change_critical_radius – The maximum relative change of the critical radius [-]
set_max_rel_change_nucleation_rate_log \( (\text{max} \_\text{rel} \_\text{change} \_\text{nucleation} \_\text{rate} \_\text{log}: \text{float} = 0.5) \)
This parameter ensures accuracy for the evolution of effective nucleation rate. **Default**: 0.5

**Parameters** max_rel_change_nucleation_rate_log – The maximum logarithmic relative change of the nucleation rate [-]

set_max_rel_radius_change \( (\text{max} \_\text{rel} \_\text{radius} \_\text{change}: \text{float} = 0.01) \)
The maximum value allowed for relative radius change in one time step. **Default**: 0.01

**Parameters** max_rel_radius_change – The maximum relative radius change in one time step [-]

set_max_rel_solute_composition_change \( (\text{max} \_\text{rel} \_\text{solute} \_\text{composition} \_\text{change}: \text{float} = 0.01) \)
Set a limit on the time step by controlling solute depletion or saturation, especially at isothermal stage. **Default**: 0.01

**Parameters** max_rel_solute_composition_change – The limit for the relative solute composition change [-]

set_max_time_step \( (\text{max} \_\text{time} \_\text{step}: \text{float} = 0.1) \)
The maximum time step allowed for time integration as fraction of the simulation time. **Default**: 0.1

**Parameters** max_time_step – The maximum time step as fraction of the simulation time [-]

set_max_time_step_during_heating \( (\text{max} \_\text{time} \_\text{step} \_\text{during} \_\text{heating}: \text{float} = 1.0) \)
The upper limit of the time step that has been enforced in the heating stages. **Default**: 1.0 s

**Parameters** max_time_step_during_heating – The maximum time step during heating [s]

set_max_volume_fraction_dissolve_time_step \( (\text{max} \_\text{volume} \_\text{fraction} \_\text{dissolve} \_\text{time} \_\text{step}: \text{float} = 0.01) \)
Sets the maximum volume fraction of subcritical particles allowed to dissolve in one time step. **Default**: 0.01

**Parameters** max_volume_fraction_dissolve_time_step – The maximum volume fraction of subcritical particles allowed to dissolve in one time step [-]

set_min_radius_nucleus_as_particle \( (\text{min} \_\text{radius} \_\text{nucleus} \_\text{as} \_\text{particle}: \text{float} = 5e-10) \)
The cut-off lower limit of precipitate radius. **Default**: 5.0E-10 m

**Parameters** min_radius_nucleus_as_particle – The minimum radius of a nucleus to be considered as a particle [m]

set_min_radius_points_per_magnitude \( (\text{min} \_\text{radius} \_\text{points} \_\text{per} \_\text{magnitude}: \text{float} = 100.0) \)
Sets the minimum number of grid points over one order of magnitude in radius. **Default**: 100.0

**Parameters** min_radius_points_per_magnitude – The minimum number of grid points over one order of magnitude in radius [-]

set_radius_points_per_magnitude \( (\text{radius} \_\text{points} \_\text{per} \_\text{magnitude}: \text{float} = 150.0) \)
Sets the number of grid points over one order of magnitude in radius. **Default**: 150.0

**Parameters** radius_points_per_magnitude – The number of grid points over one order of magnitude in radius [-]

set_rel_radius_change_class_collision \( (\text{rel} \_\text{radius} \_\text{change} \_\text{class} \_\text{collision}: \text{float} = 0.5) \)
Sets the relative radius change for avoiding class collision. **Default**: 0.5

**Parameters** rel_radius_change_class_collision – The relative radius change for avoiding class collision [-]
class tc_python.precipitation.ParticleSizeDistribution
    Bases: object
    Represents the state of a microstructure evolution at a certain time including its particle size distribution, composition and overall phase fraction.

    add_radius_and_number_density(radius: float, number_density: float)
    Adds a radius and number density pair to the particle size distribution.
    Parameters
    • radius – The radius [m]
    • number_density – The number of particles per unit volume per unit length [m^-4]
    Returns This ParticleSizeDistribution object

set_initial_composition(element_name: str, composition_value: float)
Sets the initial precipitate composition.
    Parameters
    • element_name – The name of the element
    • composition_value – The composition value [composition unit defined for the calculation]
    Returns This ParticleSizeDistribution object

Sets the type of the phase fraction or percentage. Default: By default volume fraction is used.
    Parameters volume_fraction_of_phase_type_enum – Specifies if volume percent or fraction is used
    Returns This ParticleSizeDistribution object

set_volume_fraction_of_phase_value(value: float)
Sets the overall volume fraction of the phase (unit based on the setting of set_volume_fraction_of_phase_type()).
    Parameters value – The volume fraction 0.0 - 1.0 or percent value 0 - 100
    Returns This ParticleSizeDistribution object

class tc_python.precipitation.PrecipitateElasticProperties
    Bases: object
    Represents the elastic transformation strain of a certain precipitate class.

    set_e11(e11: float)
    Sets the elastic strain tensor component e11. Default: 0.0
    Parameters e11 – The elastic strain tensor component e11
    Returns This PrecipitateElasticProperties object

    set_e12(e12: float)
    Sets the strain tensor component e12. Default: 0.0
    Parameters e12 – The elastic strain tensor component e12
    Returns This PrecipitateElasticProperties object

Note: This class is only relevant if the option TransformationStrainCalculationOption.USER_DEFINED has been chosen using PrecipitatePhase.set_transformation_strain_calculation_option(). The elastic strain can only be considered for non-spherical precipitates.
Parameters e12 – The elastic strain tensor component e12

Returns This PrecipitateElasticProperties object

set_e13 (e13: float)
Sets the elastic strain tensor component e13. Default: 0.0

Parameters e13 – The elastic strain tensor component e13

Returns This PrecipitateElasticProperties object

set_e22 (e22: float)
Sets the elastic strain tensor component e22. Default: 0.0

Parameters e22 – The elastic strain tensor component e22

Returns This PrecipitateElasticProperties object

set_e23 (e23: float)
Sets the elastic strain tensor component e23. Default: 0.0

Parameters e23 – The elastic strain tensor component e23

Returns This PrecipitateElasticProperties object

set_e33 (e33: float)
Sets the elastic strain tensor component e33. Default: 0.0

Parameters e33 – The elastic strain tensor component e33

Returns This PrecipitateElasticProperties object

class tc_python.precipitation.PrecipitateMorphology (value)
Bases: enum.Enum

Available precipitate morphologies.

CUBOID = 3
Cuboidal precipitates, only available for bulk nucleation.

NEEDLE = 1
Needle-like precipitates, only available for bulk nucleation.

PLATE = 2
Plate-like precipitates, only available for bulk nucleation.

SPHERE = 0
Spherical precipitates, this is the default morphology.

class tc_python.precipitation.PrecipitatePhase (precipitate_phase_name: str)
Bases: object

Represents a certain precipitate class (i.e. a group of precipitates with the same phase and settings).

disable_calculate_aspect_ratio_from_elastic_energy ()
Disables the automatic calculation of the aspect ratio from the elastic energy of the phase.

Returns This PrecipitatePhase object

Note: If you use this method, you are required to set the aspect ratio explicitly using the method set_aspect_ratio_value().

Default: This is the default setting (with an aspect ratio of 1.0).
disable_driving_force_approximation()

Disables driving force approximation for this precipitate class. Default: Driving force approximation is disabled.

Returns This PrecipitatePhase object

enable_calculate_aspect_ratio_from_elastic_energy()

Enables the automatic calculation of the aspect ratio from the elastic energy of the phase. Default: The aspect ratio is set to a value of 1.0.

Returns This PrecipitatePhase object

enable_driving_force_approximation()

Enables driving force approximation for this precipitate class. This approximation is often required when simulating precipitation of multiple particles that use the same phase description. E.g. simultaneous precipitation of a Metal-Carbide(MC) and Metal-Nitride(MN) if configured as different composition sets of the same phase FCC_A1. Default: Driving force approximation is disabled.

Returns This PrecipitatePhase object

Tip: Use this if simulations with several compositions sets of the same phase cause problems.

set_alias(alias: str)

Sets an alias string that can later be used to get values from a calculated result. Typically used when having the same phase for several precipitates, but with different nucleation sites. For example two precipitates of the phase M7C3 with nucleation sites in ‘Bulk’ and at ‘Dislocations’. The alias can be used instead of the phase name when retrieving simulated results.

Parameters alias – The alias string for this class of precipitates

Returns This PrecipitatePhase object

Note: Typically used when having using the same precipitate phase, but with different settings in the same calculation.

set_aspect_ratio_value(aspect_ratio_value: float)

Sets the aspect ratio of the phase. Default: An aspect ratio of 1.0.

Parameters aspect_ratio_value – The aspect ratio value

Returns This PrecipitatePhase object

Note: Only relevant if disable_calculate_aspect_ratio_from_elastic_energy() is used (which is the default).

set_gibbs_energy_addition(gibbs_energy_addition: float)

Sets a Gibbs energy addition to the Gibbs energy of the phase. Default: 0.0 J/mol

Parameters gibbs_energy_addition – The Gibbs energy addition [J/mol]

Returns This PrecipitatePhase object

set_interfacial_energy(interfacial_energy: float)

Sets the interfacial energy. Default: If the interfacial energy is not set, it is automatically calculated using a broken-bond model.

Parameters interfacial_energy – The interfacial energy [J/m^2]
Returns This PrecipitatePhase object

Note: The calculation of the interfacial energy using a broken-bond model is based on the assumption of an interface between a bcc- and a fcc-crystal structure with (110) and (111) lattice planes regardless of the actual phases.

set_interfacial_energy_estimation_prefactor (interfacial_energy_estimation_prefactor: float)
Sets the interfacial energy prefactor. Default: Prefactor of 1.0 (only relevant if the interfacial energy is automatically calculated).

Parameters interfacial_energy_estimation_prefactor – The prefactor for the calculated interfacial energy

Returns This PrecipitatePhase object

Note: The interfacial energy prefactor is an amplification factor for the automatically calculated interfacial energy. Example: interfacial_energy_estimation_prefactor = 2.5 => 2.5 * calculated interfacial energy

set_molar_volume (volume: float)
Sets the molar volume of the precipitate phase. Default: The molar volume obtained from the database. If no molar volume information is present in the database, a value of 7.0e-6 m^3/mol is used.

Parameters volume – The molar volume [m^3/mol]

Returns This PrecipitatePhase object

set_nucleation_at_dislocations (number_density=- 1)
Activates nucleation at dislocations for this class of precipitates. Calling the method overrides any other nucleation setting for this class of precipitates. Default: If not set, by default bulk nucleation is chosen.

Parameters number_density – Number density of nucleation sites. If not set, the value is calculated based on the matrix settings (grain size, dislocation density) [m^-3].

Returns This PrecipitatePhase object

set_nucleation_at_grain_boundaries (wetting_angle: float = 90.0, number_density: float = - 1)
Activates nucleation at grain boundaries for this class of precipitates. Calling the method overrides any other nucleation setting for this class of precipitates. Default: If not set, by default bulk nucleation is chosen.

Parameters

• wetting_angle – If not set, a default value of 90 degrees is used

• number_density – Number density of nucleation sites. If not set, the value is calculated based on the matrix settings (grain size) [m^-3].

Returns This PrecipitatePhase object

set_nucleation_at_grain_corners (wetting_angle: float = 90, number_density: float = - 1)
Activates nucleation at grain corners for this class of precipitates. Calling the method overrides any other nucleation setting for this class of precipitates. Default: If not set, by default bulk nucleation is chosen.

Parameters

• wetting_angle – If not set, a default value of 90 degrees is used]
**number_density** – Number density of nucleation sites. If not set, the value is calculated based on the matrix settings (grain size) \([\text{m}^{-3}]\).

**Returns** This *PrecipitatePhase* object

**set_nucleation_at_grain_edges** *(wetting_angle=90, number_density=-1)*
Activates nucleation at the grain edges for this class of precipitates. Calling the method overrides any other nucleation setting for this class of precipitates. **Default:** If not set, by default bulk nucleation is chosen.

**Parameters**

- **wetting_angle** – If not set, a default value of 90 degrees is used
- **number_density** – Number density of nucleation sites. If not set, the value is calculated based on the matrix settings (grain size) \([\text{m}^{-3}]\).

**Returns** This *PrecipitatePhase* object

**set_nucleation_in_bulk** *(number_density: float = -1.0)*
Activates nucleation in the bulk for this class of precipitates. Calling the method overrides any other nucleation setting for this class of precipitates. **Default:** This is the default setting (with an automatically calculated number density).

**Parameters** **number_density** – Number density of nucleation sites. If not set, the value is calculated based on the matrix settings (molar volume) \([\text{m}^{-3}]\)

**Returns** This *PrecipitatePhase* object

**set_phase_boundary_mobility** *(phase_boundary_mobility: float)*
Sets the phase boundary mobility. **Default:** 10.0 \([\text{m}^4/(\text{Js})]\).

**Parameters** **phase_boundary_mobility** – The phase boundary mobility \([\text{m}^4/(\text{Js})]\)

**Returns** This *PrecipitatePhase* object

**set_precipitate_morphology** *(precipitate_morphology_enum: tc_python.precipitation.PrecipitateMorphology)*
Sets the precipitate morphology. **Default:** *PrecipitateMorphology.SPHERE*

**Parameters** **precipitate_morphology_enum** – The precipitate morphology

**Returns** This *PrecipitatePhase* object

**set_transformation_strain_calculation_option** *(transformation_strain_calculation_option_enum: tc_python.precipitation.TransformationStrainCalculationOption)*
Sets the transformation strain calculation option. **Default:** *TransformationStrainCalculationOption.DISREGARD*.

**Parameters** **transformation_strain_calculation_option_enum** – The chosen option

**Returns** This *PrecipitatePhase* object

**with_elastic_properties** *(elastic_properties: tc_python.precipitation.PrecipitateElasticProperties)*
Sets the elastic properties. **Default:** The elastic transformation strain is disregarded by default.

**Parameters** **elastic_properties** – The elastic properties object

**Returns** This *PrecipitatePhase* object

**Note:** This method has only an effect if the option *TransformationStrainCalculationOption.USER_DEFINED* is chosen using the method *set_transformation_strain_calculation_option()*.

**with_growth_rate_model** *(growth_rate_model_enum: tc_python.precipitation.GrowthRateModel)*
Sets the growth rate model for the class of precipitates. **Default:** *GrowthRateModel.SIMPLIFIED*
Parameters `growth_rate_model_enum` – The growth rate model

Returns This `PrecipitatePhase` object

`with_particle_size_distribution`(*particle_size_distribution: tc_python.precipitation.ParticleSizeDistribution*)

Sets the initial particle size distribution for this class of precipitates. Default: If the initial particle size distribution is not explicitly provided, the simulation will start from a supersaturated matrix.

Parameters `particle_size_distribution` – The initial particle size distribution object

Returns This `PrecipitatePhase` object

Tip: Use this option if you want to study the further evolution of an existing microstructure.

class `tc_python.precipitation.PrecipitationCCTCalculation`(*calculation*)

Bases: `tc_python.abstract_base.AbstractCalculation`

Configuration for a Continuous-Cooling-Time (CCT) precipitation calculation.

`calculate`(*timeout_in_minutes: float = 0.0*) → `tc_python.precipitation.PrecipitationCalculationTTTorCCTResult`

Runs the CCT diagram calculation.

Parameters `timeout_in_minutes` – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than `timeout_in_minutes`, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

Returns A `PrecipitationCalculationTTTorCCTResult` which later can be used to get specific values from the calculated result

`get_system_data`() → `tc_python.abstract_base.SystemData`

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

Note: Parameters can only be read from unencrypted (i.e. `user`) databases loaded as `*.tdb`-file.

Returns The system data

`set_composition`(*element_name: str, value: float*)

Sets the composition of the elements. The unit for the composition can be changed using `set_composition_unit()`. Default: Mole percent (`CompositionUnit.MOLE_PERCENT`)

Parameters

- `element_name` – The element
- `value` – The composition (fraction or percent depending on the composition unit)

Returns This `PrecipitationCCTCalculation` object

`set_composition_unit`(*unit_enum: tc_python.utils.CompositionUnit*)

Sets the composition unit. Default: Mole percent (`CompositionUnit.MOLE_PERCENT`).

Parameters `unit_enum` – The new composition unit

Returns This `PrecipitationCCTCalculation` object

`set_cooling_rates`(*cooling_rates: List[float]*)

Sets all cooling rates for which the CCT diagram should be calculated.
Parameters `cooling_rates` – A list of cooling rates [K/s]

Returns This `PrecipitationCCTCalculation` object

`set_max_temperature`(max_temperature: float)
Sets maximum temperature of the CCT diagram.

Parameters `max_temperature` – the maximum temperature [K]

Returns This `PrecipitationCCTCalculation` object

`set_min_temperature`(min_temperature: float)
Sets the minimum temperature of the CCT diagram.

Parameters `min_temperature` – the minimum temperature [K]

Returns This `PrecipitationCCTCalculation` object

`stop_at_volume_fraction_of_phase`(stop_criterion_value: float)
Sets the stop criterion as a volume fraction of the phase. This setting is applied to all phases.

Parameters `stop_criterion_value` – the volume fraction of the phase (a value between 0 and 1)

Returns This `PrecipitationCCTCalculation` object

`with_matrix_phase`(matrix_phase: tc_python.precipitation.MatrixPhase)
Sets the matrix phase.

Parameters `matrix_phase` – The matrix phase

Returns This `PrecipitationCCTCalculation` object

`with_numerical_parameters`(numerical_parameters: tc_python.precipitation.NumericalParameters)
Sets the numerical parameters. If not specified, reasonable defaults are be used.

Parameters `numerical_parameters` – The parameters

Returns This `PrecipitationCCTCalculation` object

`with_system_modifications`(system_modifications: tc_python.abstract_base.SystemModifications)
Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Parameters `system_modifications` – The system modification to be performed

Returns This `PrecipitationCCTCalculation` object

---

Note: This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a *.tdb*-file.

---

**class** `tc_python.precipitation.PrecipitationCalculationResult`(result)
Bases: `tc_python.abstract_base.AbstractResult`

Result of a precipitation calculation. This can be used to query for specific values.

`save_to_disk`(path: str)
Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with `load_result_from_disk()`

Parameters `path` – the path to the folder you want the result to be saved in. It can be relative or absolute.
Returns this \texttt{PrecipitationCalculationResult} object

class \texttt{tc\_python.precipitation.PrecipitationCalculationSingleResult}(\texttt{result})
Bases: \texttt{tc\_python.precipitation.PrecipitationCalculationResult}

Result of a isothermal or non-isothermal precipitation calculation. This can be used to query for specific values.

Search the Thermo-Calc help for definitions of the axis variables, e.g. search isothermal variables or non-isothermal variables.

\texttt{get\_aspect\_ratio\_distribution\_for\_particle\_length\_of}(\texttt{precipitate\_id}: \texttt{str}, \texttt{time}: \texttt{float}) \to \texttt{[typing.List[float], typing.List[float]]}

Returns the aspect ratio distribution of a precipitate in dependency of its mean particle length at a certain time.

Only available if the morphology is set to \texttt{PrecipitateMorphology.NEEDLE} or \texttt{PrecipitateMorphology.PLATE}.

Parameters

- \texttt{time} – The time [s]
- \texttt{precipitate\_id} – The id of a precipitate can either be the phase name or an alias

\texttt{get\_aspect\_ratio\_distribution\_for\_radius\_of}(\texttt{precipitate\_id}: \texttt{str}, \texttt{time}: \texttt{float}) \to \texttt{[typing.List[float], typing.List[float]]}

Returns the aspect ratio distribution of a precipitate in dependency of its mean radius at a certain time.

Only available if the morphology is set to \texttt{PrecipitateMorphology.NEEDLE} or \texttt{PrecipitateMorphology.PLATE}.

Parameters

- \texttt{time} – The time [s]
- \texttt{precipitate\_id} – The id of a precipitate can either be the phase name or an alias

\texttt{get\_critical\_radius\_of}(\texttt{precipitate\_id}: \texttt{str}) \to \texttt{[typing.List[float], typing.List[float]]}

Returns the critical radius of a precipitate in dependency of the time.

Parameters \texttt{precipitate\_id} – The id of a precipitate can either be phase name or alias

Returns A tuple of two lists of floats (time [s], critical radius [m])

\texttt{get\_cubic\_factor\_distribution\_for\_particle\_length\_of}(\texttt{precipitate\_id}: \texttt{str}, \texttt{time}: \texttt{float}) \to \texttt{[typing.List[float], typing.List[float]]}

Returns the cubic factor distribution of a precipitate in dependency of its mean particle length at a certain time.

Only available if the morphology is set to \texttt{PrecipitateMorphology.CUBOID}.

Parameters

- \texttt{time} – The time in seconds
- \texttt{precipitate\_id} – The id of a precipitate can either be the phase name or an alias

Returns A tuple of two lists of floats (particle length [m], cubic factor)
get_cubic_factor_distribution_for_radius_of(precipitate_id: str, time: float) \rightarrow \text{[typing.List[float], typing.List[float]]}

Returns the cubic factor distribution of a precipitate in dependency of its mean radius at a certain time. Only available if the morphology is set to PrecipitateMorphology.CUBOID.

Parameters

- **time** – The time [s]
- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns A tuple of two lists of floats (radius [m], cubic factor)

get_driving_force_of(precipitate_id: str) \rightarrow \text{[typing.List[float], typing.List[float]]}

Returns the (by R * T) normalized driving force of a precipitate in dependency of the time.

Parameters **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns A tuple of two lists of floats (time [s], normalized driving force)

get_grain_critical_radius() \rightarrow \text{[typing.List[float], typing.List[float]]}

Returns the critical radius of grains in dependency of the time.

Returns A tuple of two lists of floats (time [s], critical radius [m])

get_grain_mean_radius() \rightarrow \text{[typing.List[float], typing.List[float]]}

Returns the mean grain size of the matrix phase in dependency of the time.

Returns A tuple of two lists of floats (time [s], mean radius [m])

get_grain_number_density() \rightarrow \text{[typing.List[float], typing.List[float]]}

Returns the grain number density in dependency of the time.

Returns A tuple of two lists of floats (time [s], grain number density [m^-3])

get_grain_number_density_distribution_for_length(time: float) \rightarrow \text{[typing.List[float], typing.List[float]]}

Returns the number density distribution of grains in dependency of its mean particle length at a certain time.

Parameters **time** – The time [s]

Returns A tuple of two lists of floats (grain length[m], number of grains per unit volume per unit length [m^-4])

get_grain_number_density_distribution_for_radius(time: float) \rightarrow \text{[typing.List[float], typing.List[float]]}

Returns the number density distribution of a grains in dependency of its mean radius at a certain time.

Parameters **time** – The time [s]

Returns A tuple of two lists of floats (radius [m], number of grains per unit volume per unit length [m^-4])

get_grain_size_distribution(time: float) \rightarrow \text{[typing.List[float], typing.List[float]]}

Returns the size distribution of the matrix phase in dependency of its grain radius length at a certain time.

Parameters **time** – The time [s]

Returns A tuple of two lists of floats (grain radius[m], number density of grains[m^-3])
get_matrix_composition_in_mole_fraction_of\( (element\_name: \text{str}) \rightarrow \text{typing.List[float], typing.List[float]} \)

Returns the matrix composition (as mole fractions) of a certain element in dependency of the time.

**Parameters**
- `element_name` – The element

**Returns**
- A tuple of two lists of floats (time [s], mole fraction)

get_matrix_composition_in_weight_fraction_of\( (element\_name: \text{str}) \rightarrow \text{typing.List[float], typing.List[float]} \)

Returns the matrix composition (as weight fraction) of a certain element in dependency of the time.

**Parameters**
- `element_name` – The element

**Returns**
- A tuple of two lists of floats (time [s], weight fraction)

get_mean_aspect_ratio_of\( (precipitate\_id: \text{str}) \rightarrow \text{typing.List[float], typing.List[float]} \)

Returns the mean aspect ratio of a precipitate in dependency of the time. Only available if the morphology is set to `PrecipitateMorphology.NEEDLE` or `PrecipitateMorphology.PLATE`.

**Parameters**
- `precipitate_id` – The id of a precipitate can either be the phase name or an alias

**Returns**
- A tuple of two lists of floats (time [s], mean aspect ratio)

get_mean_cubic_factor_of\( (precipitate\_id: \text{str}) \rightarrow \text{typing.List[float], typing.List[float]} \)

Returns the mean cubic factor of a precipitate in dependency of the time. Only available if the morphology is set to `PrecipitateMorphology.CUBOID`.

**Parameters**
- `precipitate_id` – The id of a precipitate can either be the phase name or an alias

**Returns**
- A tuple of two lists of floats (time [s], mean cubic factor)

get_mean_particle_length_of\( (precipitate\_id: \text{str}) \rightarrow \text{typing.List[float], typing.List[float]} \)

Returns the mean particle length of a precipitate in dependency of the time. Only available if the morphology is set to `PrecipitateMorphology.NEEDLE` or `PrecipitateMorphology.PLATE`.

**Parameters**
- `precipitate_id` – The id of a precipitate can either be the phase name or an alias

**Returns**
- A tuple of two lists of floats (time [s], mean particle length [m])

get_mean_radius_of\( (precipitate\_id: \text{str}) \rightarrow \text{typing.List[float], typing.List[float]} \)

Returns the mean radius of a precipitate in dependency of the time.

**Parameters**
- `precipitate_id` – The id of a precipitate can either be the phase name or alias

**Returns**
- A tuple of two lists of floats (time [s], mean radius [m])

get_normalized_grain_size_distribution\( (time: \text{float}) \rightarrow \text{typing.List[float], typing.List[float]} \)

Returns the normalized number density distribution of a grains at a certain time.

**Parameters**
- `time` – The time [s]

**Returns**
- A tuple of two lists of floats (Normalized size, Frequency)

get_normalized_number_density_distribution_of\( (precipitate\_id: \text{str}, time: \text{float}) \rightarrow \text{typing.List[float], typing.List[float]} \)

Returns the normalized number density distribution of a precipitate at a certain time.

**Parameters**
- `precipitate_id` – The id of a precipitate
- `time` – The time [s]

**Returns**
- A tuple of two lists of floats (Normalized size, Frequency)
Parameters

- **time** – The time [s]
- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns A tuple of two lists of floats (Normalized size, Frequency)

get_nucleation_rate_of \((\text{precipitate_id}: \text{str}) \rightarrow [\text{typing.List[float]}, \text{typing.List[float]}]\)

Returns the nucleation rate of a precipitate in dependency of the time.

Parameters **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns A tuple of two lists of floats (time [s], nucleation rate \([\text{m}^{-3} \cdot \text{s}^{-1}]\))

get_number_density_distribution_for_particle_length_of \((\text{precipitate_id}: \text{str}, \text{time}: \text{float}) \rightarrow [\text{typing.List[float]}, \text{typing.List[float]}]\)

Returns the number density distribution of a precipitate in dependency of its mean particle length at a certain time.

Parameters

- **time** – The time [s]
- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns A tuple of two lists of floats (particle length[m], number of particles per unit volume per unit length \([\text{m}^{-4}]\))

get_number_density_distribution_for_radius_of \((\text{precipitate_id}: \text{str}, \text{time}: \text{float}) \rightarrow [\text{typing.List[float]}, \text{typing.List[float]}]\)

Returns the number density distribution of a precipitate in dependency of its mean radius at a certain time.

Parameters

- **time** – The time [s]
- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns A tuple of two lists of floats (radius [m], number of particles per unit volume per unit length \([\text{m}^{-4}]\))

get_number_density_of \((\text{precipitate_id}: \text{str}) \rightarrow [\text{typing.List[float]}, \text{typing.List[float]}]\)

Returns the particle number density of a precipitate in dependency of the time.

Parameters **precipitate_id** – The id of a precipitate can either be phase name or alias

Returns A tuple of two lists of floats (time [s], particle number density \([\text{m}^{-3}]\))

get_precipitate_composition_in_mole_fraction_of \((\text{precipitate_id}: \text{str}, \text{element_name}: \text{str}) \rightarrow [\text{typing.List[float]}, \text{typing.List[float]}]\)

Returns the precipitate composition (as mole fractions) of a certain element in dependency of the time.

Parameters

- **precipitate_id** – The id of a precipitate can either be phase name or alias
- **element_name** – The element

Returns A tuple of two lists of floats (time [s], mole fraction)
get_precipitate_composition_in_weight_fraction_of (precipitate_id: str, element_name: str) → [typing.List[float], typing.List[float]]

Returns the precipitate composition (as weight fraction) of a certain element in dependency of the time.

**Parameters**

- **precipitate_id** – The id of a precipitate can either be phase name or alias
- **element_name** – The element

**Returns** A tuple of two lists of floats (time [s], weight fraction)

get_size_distribution_for_particle_length_of (precipitate_id: str, time: float) → [typing.List[float], typing.List[float]]

Returns the size distribution of a precipitate in dependency of its mean particle length at a certain time.

**Parameters**

- **time** – The time [s]
- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

**Returns** A tuple of two lists of floats (particle length[m], number of particles per unit volume per unit length [m^-4])

get_size_distribution_for_radius_of (precipitate_id: str, time: float) → [typing.List[float], typing.List[float]]

Returns the size distribution of a precipitate in dependency of its mean radius at a certain time.

**Parameters**

- **time** – The time [s]
- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

**Returns** A tuple of two lists of floats (radius [m], number of particles per unit volume per unit length [m^-4])

get_volume_fraction_of (precipitate_id: str) → [typing.List[float], typing.List[float]]

Returns the volume fraction of a precipitate in dependency of the time.

**Parameters**

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

**Returns** A tuple of two lists of floats (time [s], volume fraction)

class tc_python.precipitation.PrecipitationCalculationTTTOrCCTResult (result)
Bases: tc_python.precipitation.PrecipitationCalculationCalculationResult

Result of a TTT or CCT precipitation calculation.

get_result_for_precipitate (precipitate_id: str) → [typing.List[float], typing.List[float]]

Returns the calculated data of a TTT or CCT diagram for a certain precipitate.

**Parameters**

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

**Returns** A tuple of two lists of floats (time [s], temp [K])

class tc_python.precipitation.PrecipitationIsoThermalCalculation (calculation)
Bases: tc_python.abstract_base.AbstractCalculation

Configuration for an isothermal precipitation calculation.
calculate (timeout_in_minutes: float = 0.0) \(\rightarrow tc\_python\_precipitation.PrecipitationCalculationSingleResult\)

Runs the isothermal precipitation calculation.

**Parameters**

- **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

**Returns**

A PrecipitationCalculationSingleResult which later can be used to get specific values from the calculated result.

get_system_data () \(\rightarrow tc\_python\_abstract\_base.SystemData\)

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using with_system_modifications().

**Note:** Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.*

**Returns**

The system data.

set_composition (element_name: str, value: float)

Sets the composition of the elements. The unit for the composition can be changed using set_composition_unit(). Default: Mole percent (CompositionUnit.MOLE_PERCENT)

**Parameters**

- **element_name** – The element
- **value** – The composition (fraction or percent depending on the composition unit)

**Returns**

This PrecipitationIsoThermalCalculation object.

set_composition_unit (unit_enum: tc_python.utils.CompositionUnit = <CompositionUnit.MOLE_PERCENT: 2>)

Sets the composition unit. Default: Mole percent (CompositionUnit.MOLE_PERCENT).

**Parameters**

- **unit_enum** – The new composition unit

**Returns**

This PrecipitationIsoThermalCalculation object.

set_simulation_time (simulation_time: float)

Sets the simulation time.

**Parameters**

- **simulation_time** – The simulation time [s]

**Returns**

This PrecipitationIsoThermalCalculation object.

set_temperature (temperature: float)

Sets the temperature for the isothermal simulation.

**Parameters**

- **temperature** – the temperature [K]

**Returns**

This PrecipitationIsoThermalCalculation object.

with_matrix_phase (matrix_phase: tc_python.precipitation.MatrixPhase)

Sets the matrix phase.

**Parameters**

- **matrix_phase** – The matrix phase

**Returns**

This PrecipitationIsoThermalCalculation object.

with_numerical_parameters (numerical_parameters: tc_python.precipitation.NumericalParameters)

Sets the numerical parameters. If not specified, reasonable defaults are be used.
Parameters `numerical_parameters` – The parameters

Returns This `PrecipitationIsoThermalCalculation` object

`with_system_modifications` *(system_modifications: tc_python.abstract_base.SystemModifications)*

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

**Note:** This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a `.tdb`-file.

Parameters `system_modifications` – The system modification to be performed

Returns This `PrecipitationIsoThermalCalculation` object

---

```python
class tc_python.precipitation.PrecipitationNonIsoThermalCalculation(calculation)
Bases: tc_python.abstract_base.AbstractCalculation
```

Configuration for a non-isothermal precipitation calculation.

**calculate** *(timeout_in_minutes: float = 0.0) → tc_python.precipitation.PrecipitationCalculationSingleResult*

Runs the non-isothermal precipitation calculation.

Parameters `timeout_in_minutes` – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

Returns A `PrecipitationCalculationSingleResult` which later can be used to get specific values from the calculated result

```python
get_system_data () → tc_python.abstract_base.SystemData
```

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

**Note:** Parameters can only be read from unencrypted (i.e. user) databases loaded as `.tdb`-file.

Returns The system data

```python
set_composition (element_name: str, value: float)
```

Sets the composition of the elements. The unit for the composition can be changed using `set_composition_unit()`. Default: Mole percent (`CompositionUnit.MOLE_PERCENT`)

Parameters

- `element_name` – The element
- `value` – The composition (fraction or percent depending on the composition unit)

Returns This `PrecipitationIsoThermalCalculation` object

```python
set_composition_unit (unit_enum: tc_python.utils.CompositionUnit)
```

Sets the composition unit. Default: Mole percent (`CompositionUnit.MOLE_PERCENT`).

Parameters `unit_enum` – The new composition unit

Returns This `PrecipitationIsoThermalCalculation` object
**set_simulation_time** (*simulation_time: float*)

Sets the simulation time.

Parameters **simulation_time** – The simulation time [s]

Returns This PrecipitationNonThermalCalculation object

**with_matrix_phase** (*matrix_phase: tc_python.precipitation.MatrixPhase*)

Sets the matrix phase.

Parameters **matrix_phase** – The matrix phase

Returns This PrecipitationIsoThermalCalculation object

**with_numerical_parameters** (*numerical_parameters: tc_python.precipitation.NumericalParameters*)

Sets the numerical parameters. If not specified, reasonable defaults are be used.

Parameters **numerical_parameters** – The parameters

Returns This PrecipitationIsoThermalCalculation object

**with_system_modifications** (*system_modifications: tc_python.abstract_base.SystemModifications*)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a *.tdb*-file.

Parameters **system_modifications** – The system modification to be performed

Returns This PrecipitationNonThermalCalculation object

**with_temperature_profile** (*temperature_profile: tc_python.utils.TemperatureProfile*)

Sets the temperature profile to use with this calculation.

Parameters **temperature_profile** – the temperature profile object (specifying time / temperature points)

Returns This PrecipitationNonThermalCalculation object

**class tc_python.precipitation.PrecipitationTTTCalculation** (*calculation*)

**Bases:** tc_python.abstract_base.AbstractCalculation

Configuration for a TTT (Time-Temperature-Transformation) precipitation calculation.

**calculate** (*timeout_in_minutes: float = 0.0*) → tc_python.precipitation.PrecipitationCalculationTTToCCTResult

Runs the TTT diagram calculation.

Parameters **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

Returns A PrecipitationCalculationTTToCCTResult which later can be used to get specific values from the calculated result.

**get_system_data** () → tc_python.abstract_base.SystemData

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using with_system_modifications().
Note: Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.

Returns The system data

```python
set_composition(element_name: str, value: float)
```
Sets the composition of the elements. The unit for the composition can be changed using `set_composition_unit()`. Default: Mole percent (CompositionUnit.MOLE_PERCENT)

Parameters

- **element_name** – The element
- **value** – The composition (fraction or percent depending on the composition unit)

Returns This `PrecipitationTTTCalculation` object

```python
set_composition_unit(unit_enum: tc_python.utils.CompositionUnit)
```
Sets the composition unit. Default: Mole percent (CompositionUnit.MOLE_PERCENT).

Parameters unit_enum – The new composition unit

Returns This `PrecipitationTTTCalculation` object

```python
set_max_annealing_time(max_annealing_time: float)
```
Sets the maximum annealing time, i.e. the maximum time of the simulation if the stopping criterion is not reached.

Parameters max_annealing_time – the maximum annealing time [s]

Returns This `PrecipitationTTTCalculation` object

```python
set_max_temperature(max_temperature: float)
```
Sets the maximum temperature for the TTT diagram.

Parameters max_temperature – the maximum temperature [K]

Returns This `PrecipitationTTTCalculation` object

```python
set_min_temperature(min_temperature: float)
```
Sets the minimum temperature for the TTT diagram.

Parameters min_temperature – the minimum temperature [K]

Returns This `PrecipitationTTTCalculation` object

```python
set_temperature_step(temperature_step: float)
```
Sets the temperature step for the TTT diagram. If not set, the default value is 10 K.

Parameters temperature_step – the temperature step [K]

Returns This `PrecipitationTTTCalculation` object

```python
stop_at_percent_of_equilibrium_fraction(percentage: float)
```
Sets the stop criterion to a percentage of the overall equilibrium phase fraction, alternatively a required volume fraction can be specified (using `stop_at_volume_fraction_of_phase()`).

Parameters percentage – the percentage to stop at (value between 0 and 100)

Returns This `PrecipitationTTTCalculation` object
**stop_at_volume_fraction_of_phase** *(volume_fraction: float)*
Sets the stop criterion as a volume fraction of the phase, alternatively a required percentage of the equilibrium phase fraction can be specified (using *stop_at_percent_of_equilibria_fraction()*). Stopping at a specified volume fraction is the default setting.

This setting is applied to all phases.

- **Parameters** **volume_fraction** – the volume fraction to stop at (a value between 0 and 1)

- **Returns** This *PrecipitationTTTCalculation* object

**with_matrix_phase** *(matrix_phase: tc_python.precipitation.MatrixPhase)*
Sets the matrix phase.

- **Parameters** **matrix_phase** – The matrix phase

- **Returns** This *PrecipitationTTTCalculation* object

**with_numerical_parameters** *(numerical_parameters: tc_python.precipitation.NumericalParameters)*
Sets the numerical parameters. If not specified, reasonable defaults are be used.

- **Parameters** **numerical_parameters** – The parameters

- **Returns** This *PrecipitationTTTCalculation* object

**with_system_modifications** *(system_modifications: tc_python.abstract_base.SystemModifications)*
Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

- **Parameters** **system_modifications** – The system modification to be performed

- **Returns** This *PrecipitationTTTCalculation* object

---

**class tc_python.precipitation.TransformationStrainCalculationOption*(value)*
Bases: enum.Enum
Options for calculating the transformation strain.

- **CALCULATE_FROM_MOLAR_VOLUME = 2**
  Calculates the transformation strain from the molar volume, obtains a purely dilatational strain.

- **DISREGARD = 1**
  Ignores the transformation strain, **this is the default setting**.

- **USER_DEFINED = 3**
  Transformation strain to be specified by the user.

**class tc_python.precipitation.VolumeFractionOfPhaseType*(value)*
Bases: enum.Enum
Unit of the volume fraction of a phase.

- **VOLUME_FRACTION = 6**
  Volume fraction (0 - 1), **this is the default**.

- **VOLUME_PERCENT = 5**
  Volume percent (0% - 100%).
5.1.4 Module “scheil”

class tc_python.scheil.CalculateSecondaryDendriteArmSpacing
   Bases: tc_python.scheil.ScheilBackDiffusion

   Configures a secondary dendrite arm spacing calculation used by Scheil \textit{with back diffusion}. The used equation is $c \times \text{cooling\_rate}^{(-n)}$ with $c$ and $n$ being provided either by the user or taken from the defaults.

   \textbf{set\_c}(c: float = 5e-05)
   
   Sets the scaling factor $c$ in the governing equation $c \times \text{cooling\_rate}^{(-n)}$.

   Default: 50 $\mu$m

   Parameters $c$ – The scaling factor [m]

   Returns This CalculateSecondaryDendriteArmSpacing object

   \textbf{set\_cooling\_rate}(cooling\_rate: float = 1.0)

   Sets the cooling rate.

   Default: 1.0 K/s

   An increased value moves the result from equilibrium toward a Scheil-Gulliver calculation.

   Parameters cooling\_rate – The cooling rate [K/s]

   Returns This CalculateSecondaryDendriteArmSpacing object

   \textbf{set\_fast\_diffusing\_elements}(element\_names: List[str])

   Sets elements as fast diffusing. This allows redistribution of these elements in both the solid and liquid parts of the alloy.

   Default: No fast-diffusing elements.

   Parameters element\_names – The elements

   Returns This CalculateSecondaryDendriteArmSpacing object

   \textbf{set\_n}(n: float = 0.33)

   Sets the exponent $n$ in the governing equation $c \times \text{cooling\_rate}^{(-n)}$.

   Default: 0.33

   Parameters $n$ – The exponent [-]

   Returns This CalculateSecondaryDendriteArmSpacing object

   \textbf{set\_primary\_phasename}(primary\_phase\_name: str = 'AUTOMATIC')

   Sets the name of the primary phase.

   The primary phase is the phase where the back diffusion takes place. If \textit{AUTOMATIC} is selected, the program tries to find the phase which will give the most back diffusion. That behavior can be overridden by selecting a specific primary phase.

   Default: \textit{AUTOMATIC}

   Parameters primary\_phase\_name – The phase name (or \textit{AUTOMATIC})

   Returns This CalculateSecondaryDendriteArmSpacing object

class tc_python.scheil.ConstantSecondaryDendriteArmSpacing
   Bases: tc_python.scheil.ScheilBackDiffusion

   Configures a constant secondary dendrite arm spacing used by Scheil \textit{with back diffusion}. The secondary dendrite arm spacing can either be provided by the user or taken from the defaults.
set_cooling_rate(cooling_rate: float = 1.0)
Sets the cooling rate.

**Default:** 1.0 K/s

An increased value moves the result from equilibrium toward a Scheil-Gulliver calculation.

**Parameters**

- **cooling_rate** – The cooling rate [K/s]

**Returns**

This `ConstantSecondaryDendriteArmSpacing` object

set_fast_diffusing_elements(element_names: List[str])
Sets elements as fast diffusing. This allows redistribution of these elements in both the solid and liquid parts of the alloy.

**Default:** No fast-diffusing elements.

**Parameters**

- **element_names** – The elements

**Returns**

This `ConstantSecondaryDendriteArmSpacing` object

set_primary_phasename(primary_phase_name: str = 'AUTOMATIC')
Sets the name of the primary phase.

The primary phase is the phase where the back diffusion takes place. If `AUTOMATIC` is selected, the program tries to find the phase which will give the most back diffusion. That behavior can be overridden by selecting a specific primary phase.

**Default:** `AUTOMATIC`

**Parameters**

- **primary_phase_name** – The phase name (or `AUTOMATIC`)

**Returns**

This `ConstantSecondaryDendriteArmSpacing` object

class tc_python.scheil.ScheilBackDiffusion
Bases: `tc_python.scheil.ScheilCalculationType`

Configuration for back diffusion in the solid primary phase.

**Warning:** This feature has only effect on systems with diffusion data (typically a mobility database). If used for a system without diffusion data, a normal Scheil calculation is done.

classmethod calculate_secondary_dendrite_arm_spacing()
Calculate the secondary dendrite arm spacing based on the following equation: \( c \times cooling_rate^{-n} \) with \( c \) and \( n \) being provided either by the user or taken from the defaults.

Use the methods provide by `CalculateSecondaryDendriteArmSpacing` to configure the parameters.

**Returns**

A `CalculateSecondaryDendriteArmSpacing` object

classmethod constant_secondary_dendrite_arm_spacing(secondary_dendrite_arm_spacing: float = 5e-05)
Assuming constant secondary dendrite arm spacing, provided either by the user or taken from the defaults.

**Default:** 50 µm

**Parameters**

- **secondary_dendrite_arm_spacing** – The dendrite arm spacing [m]

**Returns**

A `ConstantSecondaryDendriteArmSpacing` object

class tc_python.scheil.ScheilCalculation(calculator)
Bases: `tc_python.abstract_base.AbstractCalculation`
Configuration for a Scheil solidification calculation.

**Note:** Specify the settings, the calculation is performed with `calculate()`.

```python
calculate (timeout_in_minutes: float = 0.0) -> tc_python.scheil.ScheilCalculationResult
```

Runs the Scheil calculation.

**Warning:** Scheil calculations do not support the GAS phase being selected, this means the GAS phase must always be deselected in the system if it is present in the database.

**Parameters** `timeout_in_minutes` – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than `timeout_in_minutes`, a `UnrecoverableCalculationException` will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

**Returns** A `ScheilCalculationResult` which later can be used to get specific values from the simulation.

```python
disable_global_minimization ()
```

Disables global minimization.

**Default:** Enabled

**Note:** When enabled, a global minimization test is performed when an equilibrium is reached. This costs more computer time but the calculations are more robust.

**Returns** This `ScheilCalculation` object

```python
enable_global_minimization()  
```

Enables global minimization.

**Default:** Enabled

**Note:** When enabled, a global minimization test is performed when an equilibrium is reached. This costs more computer time but the calculations are more robust.

**Returns** This `ScheilCalculation` object

```python
get_system_data () -> tc_python.abstract_base.SystemData
```

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

**Note:** Parameters can only be read from unencrypted (i.e. `user`) databases loaded as *.tdb-file.

**Returns** The system data

---

5.1. Calculations
**set_composition** *(component_name: str, value: float)*

Sets the composition of a component. The unit for the composition can be changed using `set_composition_unit()`.

**Default:** Mole percent (`CompositionUnit.MOLE_PERCENT`)

**Parameters**

- **component_name** – The component
- **value** – The composition value [composition unit defined for the calculation]

**Returns** This `ScheilCalculation` object

**set_composition_unit** *(unit_enum: tc_python.utils.CompositionUnit = CompositionUnit.MOLE_PERCENT)*

Sets the composition unit.

**Default:** Mole percent (`CompositionUnit.MOLE_PERCENT`).

**Parameters**

- **unit_enum** – The new composition unit

**Returns** This `ScheilCalculation` object

**set_start_temperature** *(temperature_in_kelvin: float = 2500.0)*

Sets the start temperature.

**Warning:** The start temperature needs to be higher than the liquidus temperature of the alloy.

**Default:** 2500.0 K

**Parameters**

- **temperature_in_kelvin** – The temperature [K]

**Returns** This `ScheilCalculation` object

**with_calculation_type** *(scheil_calculation_type: tc_python.scheil.ScheilCalculationType)*

Chooses a specific Scheil calculation. ClassicScheil for only setting fast diffusers, ScheilBackDiffusion enables back diffusion in the solid primary phase and optionally fast diffusers in all solid phases, and ScheilSoluteTrapping enables solute trapping in the solid primary phase.

**Parameters**

- **scheil_type** – Type of Scheil calculation, either ScheilClassic, ScheilBackDiffusion or ScheilSoluteTrapping

**Returns** This `ScheilCalculation` object

**with_options** *(options: tc_python.scheil.ScheilOptions)*

Sets the Scheil simulation options.

**Parameters**

- **options** – The Scheil simulation options

**Returns** This `ScheilCalculation` object

**with_system_modifications** *(system_modifications: tc_python.abstract_base.SystemModifications)*

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

**Note:** This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a *.tdb*-file.

**Parameters**

- **system_modifications** – The system modification to be performed

**Returns** This `ScheilCalculation` object
class tc_python.scheil.ScheilCalculationResult(result)
    Bases: tc_python.abstract_base.AbstractResult

Result of a Scheil calculation.

get_values_grouped_by_quantity_of(x_quantity: Union[tc_python.quantity_factory.ScheilQuantity, str], y_quantity: Union[tc_python.quantity_factory.ScheilQuantity, str], sort_and_merge: bool = True) → Dict[str, tc_python.utils.ResultValueGroup]

Returns x-y-line data grouped by the multiple datasets of the specified quantities (for example dependency of phases or components). Use get_values_of() instead if you need no separation. The available quantities can be found in the documentation of the factory class ScheilQuantity.

Note: The different datasets might contain NaN-values between different subsections and might not be sorted even if the flag `sort_and_merge` has been set (because they might be unsortable due to their nature).

Parameters
- **x_quantity** – The first Scheil quantity (“x-axis”), Console Mode syntax strings can be used as an alternative (for example “T”)
- **y_quantity** – The second Scheil quantity (“y-axis”), Console Mode syntax strings can be used as an alternative (for example “NV”)
- **sort_and_merge** – If True, the data is sorted and merged into as few subsections as possible (divided by NaN)

Returns Containing the ResultValueGroup dataset objects with their quantity labels as keys

get_values_grouped_by_stable_phases_of(x_quantity: Union[tc_python.quantity_factory.ScheilQuantity, str], y_quantity: Union[tc_python.quantity_factory.ScheilQuantity, str], sort_and_merge: bool = True) → Dict[str, tc_python.utils.ResultValueGroup]

Returns x-y-line data grouped by the sets of “stable phases” (for example “LIQUID” or “LIQUID + FCC_A1”). Use get_values_of() instead if you need no separation. The available quantities can be found in the documentation of the factory class ScheilQuantity.

Note: The different datasets might contain NaN-values between different subsections and might not be sorted even if the flag `sort_and_merge` has been set (because they might be unsortable due to their nature).

Parameters
- **x_quantity** – The first Scheil quantity (“x-axis”), Console Mode syntax strings can be used as an alternative (for example “T”)
- **y_quantity** – The second Scheil quantity (“y-axis”), Console Mode syntax strings can be used as an alternative (for example “NV”)
- **sort_and_merge** – If True, the data will be sorted and merged into as few subsections as possible (divided by NaN)
Returns Containing the ResultValueGroup dataset objects with their “stable phases” labels as keys

\[ \text{get\_values\_of}( \text{x\_quantity: Union[tc\_python.quantity\_factory.ScheilQuantity, str], y\_quantity: Union[tc\_python.quantity\_factory.ScheilQuantity, str]} ) \rightarrow [\text{typing.List}[\text{float}], \text{typing.List}[\text{float}]] \]

Returns sorted x-y-line data without any separation. Use \text{get\_values\_grouped\_by\_quantity\_of()} or \text{get\_values\_grouped\_by\_stable\_phases\_of()} instead if you need such a separation. The available quantities can be found in the documentation of the factory class ScheilQuantity.

Note: This method will always return sorted data without any NaN-values. In case of ambiguous quantities (for example: CompositionOfPhaseAsWeightFraction("FCC_A1", "All")) that can give data that is hard to interpret. In such a case you need to choose the quantity in another way or use one of the other methods.

Parameters

- **x\_quantity** – The first Scheil quantity (“x-axis”), Console Mode syntax strings can be used as an alternative (for example “T”)
- **y\_quantity** – The second Scheil quantity (“y-axis”), Console Mode syntax strings can be used as an alternative (for example “NV”)

Returns A tuple containing the x- and y-data in lists

\[ \text{save\_to\_disk}( \text{path: str} ) \]

Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with \text{load\_result\_from\_disk()}

Parameters **path** – the path to the folder you want the result to be saved in.

Returns this ScheilCalculationResult object

\[ \text{class tc\_python.scheil.ScheilCalculationType} \]

Bases: object

Specific configuration for the different Scheil calculation types

\[ \text{classmethod scheil\_back\_diffusion()} \]

Configuration for back diffusion in the solid primary phase.

Warning: This feature has only effect on systems with diffusion data (typically a mobility database). If used for a system without diffusion data, a normal Scheil calculation is done. :return: A ScheilBackDiffusion

\[ \text{classmethod scheil\_classic()} \]

Configuration for Classic Scheil with fast diffusers. :return: A ScheilClassic

\[ \text{classmethod scheil\_solute\_trapping()} \]

Configures the Scheil solute trapping settings. The used solidification speed equation is Scanning speed \* \( \cos(\text{angle}) \) with Scanning speed and angle being provided either by the user or taken from the defaults. :return: A ScheilSoluteTrapping

\[ \text{class tc\_python.scheil.ScheilClassic} \]

Bases: tc\_python.scheil.ScheilCalculationType

Configuration for Classic Scheil with fast diffusers.
set_fast_diffusing_elements(element_names: List[str])

Sets elements as fast diffusing. This allows redistribution of these elements in both the solid and liquid parts of the alloy.

**Default:** No fast-diffusing elements.

**Parameters**

- `element_names` – The elements

**Returns**

This `ScheilClassic` object

class tc_python.scheil.ScheilOptions

Bases: object

Options for the Scheil simulation.

calculate_from_gas()

Calculates the evaporation temperature if a gas phase is selected in the system, and then calculates equilibria in the gas+liquid and liquid regions until liquidus temperature is reached.

**Default:** Calculation starts from liquidus temperature.

**Returns**

This `ScheilOptions` object

calculate_from_liquidus()

Solidification calculation starting from the liquidus temperature. Liquid properties between start temperature and liquidus are not obtainable.

**Default:** Calculation starts from liquidus temperature.

**Returns**

This `ScheilOptions` object

calculate_from_start_temperature()

Calculation of equilibria from start temperature at 50 K intervals until liquidus temperature is reached. This option makes it possible to obtain properties of the liquid phase before the solidification starts.

**Default:** Calculation starts from liquidus temperature.

**Returns**

This `ScheilOptions` object

disable_approximate_driving_force_for_metastable_phases()

Disables the approximation of the driving force for metastable phases.

**Default:** Enabled

**Note:** When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

If it is important that these driving forces are correct, use `disable_approximate_driving_force_for_metastable_phases()` to force the calculation to converge for the metastable phases.

**Returns**

This `ScheilOptions` object

disable_control_step_size_during_minimization()

Disables stepsize control during minimization (non-global).

**Default:** Enabled

**Returns**

This `ScheilOptions` object
**disable_equilibrium_solidification_calculation()**
Skips the property (one axis) diagram calculation of solidification under equilibrium conditions, before the Scheil solidification calculation starts.

In general it is not necessary to perform this calculation.

**Default:** Disabled. The equilibrium solidification calculation is skipped.

**Returns**  This *ScheilOptions* object

**disable_force_positive_definite_phase_hessian()**
Disables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for “Hessian minimization”.

**Default:** Enabled

**Returns**  This *ScheilOptions* object

**enable_approximate_driving_force_for_metastable_phases()**
Enables the approximation of the driving force for metastable phases.

**Default:** Enabled

**Note:**  When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

If it is important that these driving forces are correct, use `disable_approximate_driving_force_for_metastable_phases()` to force the calculation to converge for the metastable phases.

**Returns**  This *ScheilOptions* object

**enable_control_step_size_during_minimization()**  
Enables stepsize control during normal minimization (non-global).

**Default:** Enabled

**Returns**  This *ScheilOptions* object

**enable_equilibrium_solidification_calculation()**
Performs a property (one axis) diagram calculation of solidification under equilibrium conditions, before the Scheil solidification calculation starts, in the same way as is typically done in graphical and console mode.

In general it is not necessary to perform this calculation.

**Default:** Disabled. The equilibrium solidification calculation is skipped.

**Returns**  This *ScheilOptions* object

**enable_force_positive_definite_phase_hessian()**
Enables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for “Hessian minimization”.

**Default:** Enabled

**Returns**  This *ScheilOptions* object
**set_gas_phase** *(phase_name: str = ‘GAS’)*

Sets the phase used as the gas phase.

**Default:** The phase “GAS”.

**Parameters** *phase_name* – The phase name

**Returns** This *ScheilOptions* object

**set_global_minimization_max_grid_points** *(max_grid_points: int = 2000)*

Sets the maximum number of grid points in global minimization. **Only applicable if global minimization is actually used**.

**Default:** 2000 points

**Parameters** *max_grid_points* – The maximum number of grid points

**Returns** This *ScheilOptions* object

**set_global_minimization_test_interval** *(global_test_interval: int = 10)*

Sets the interval for the global test.

**Default:** 10

**Parameters** *global_test_interval* – The global test interval

**Returns** This *ScheilOptions* object

**set_liquid_phase** *(phase_name: str = ‘LIQUID’)*

Sets the phase used as the liquid phase.

**Default:** The phase “LIQUID”.

**Parameters** *phase_name* – The phase name

**Returns** This *ScheilOptions* object

**set_max_no_of_iterations** *(max_no_of_iterations: int = 500)*

Set the maximum number of iterations.

**Default:** max. 500 iterations

**Note:** As some models give computation times of more than 1 CPU second/iteration, this number is also used to check the CPU time and the calculation stops if 500 CPU seconds/iterations are used.

**Parameters** *max_no_of_iterations* – The max. number of iterations

**Returns** This *ScheilOptions* object

**set_required_accuracy** *(accuracy: float = 1e-06)*

Sets the required relative accuracy.

**Default:** 1.0E-6

**Note:** This is a relative accuracy, and the program requires that the relative difference in each variable must be lower than this value before it has converged. A larger value normally means fewer iterations but less accurate solutions. The value should be at least one order of magnitude larger than the machine precision.

**Parameters** *accuracy* – The required relative accuracy

---

5.1. Calculations
Returns This `ScheilOptions` object

**set_smallest_fraction** *(smallest_fraction: float = 1e-12)*

Sets the smallest fraction for constituents that are unstable.

It is normally only in the gas phase that you can find such low fractions.

The default value for the smallest site-fractions is 1E-12 for all phases except for IDEAL phase with one sublattice site (such as the GAS mixture phase in many databases) for which the default value is always as 1E-30.

Parameters smallest_fraction – The smallest fraction for constituents that are unstable

Returns This `ScheilOptions` object

**set_temperature_step** *(temperature_step_in_kelvin: float = 1.0)*

Sets the temperature step. Decreasing the temperature step increases the accuracy, but the default value is usually adequate.

Default step: 1.0 K

Parameters temperature_step_in_kelvin – The temperature step [K]

Returns This `ScheilOptions` object

**terminate_on_fraction_of_liquid_phase** *(fraction_to_terminate_at: float = 0.01)*

Sets the termination condition to a specified remaining fraction of liquid phase.

Default: Terminates at 0.01 fraction of liquid phase.

Note: Either the termination criterion is set to a temperature or fraction of liquid limit, both together are not possible.

Parameters fraction_to_terminate_at – the termination fraction of liquid phase (value between 0 and 1)

Returns This `ScheilOptions` object

**terminate_on_temperature** *(temperature_in_kelvin: float)*

Sets the termination condition to a specified temperature.

Default: Terminates at 0.01 fraction of liquid phase, i.e. not at a specified temperature.

Note: Either the termination criterion is set to a temperature or fraction of liquid limit, both together are not possible.

Parameters temperature_in_kelvin – the termination temperature [K]

Returns This `ScheilOptions` object

```python
class tc_python.scheil.ScheilSoluteTrapping
    Bases: tc_python.scheil.ScheilCalculationType
```

Configures the Scheil solute trapping settings. The used solidification speed equation is \( \text{Scanning speed} \times \cos(\text{angle}) \) with \text{Scanning speed} and \text{angle} being provided either by the user or taken from the defaults.
**set_angle** *(alpha: float = 45.0)*
Sets the transformation angle alpha between the solid/liquid boundary and laser scanning direction.

**Default**: 45.0

- **Parameters** **alpha** – The transformation angle [degree]
- **Returns** This *ScheilSoluteTrapping* object

**set_primary_phasename** *(primary_phase_name: str = 'AUTOMATIC')*
Sets the name of the primary phase.

The primary phase is the phase where solute trapping takes place. A necessary condition for this phase is that the phase definition contains all of the elements that are chosen in the system. When *AUTOMATIC* is selected, the program tries to find a suitable primary phase that fills this condition.

**Default**: *AUTOMATIC*

- **Parameters** **primary_phase_name** – The phase name (or *AUTOMATIC*)
- **Returns** This *ScheilSoluteTrapping* object

**set_scanning_speed** *(scanning_speed: float = 1.0)*
Sets the scanning speed.

**Default**: 1 m/s

- **Parameters** **scanning_speed** – The scaling factor [m/s]
- **Returns** This *ScheilSoluteTrapping* object

### 5.1.5 Module “step_or_map_diagrams”

**class** *tc_python.step_or_map_diagrams.AbstractAxisType*

Bases: object

The abstract base class for all axis types.

**class** *tc_python.step_or_map_diagrams.AbstractPhaseDiagramCalculation*(calculator)

Bases: *tc_python.abstract_base.AbstractCalculation*

Abstract configuration required for a property diagram calculation.

---

**Note**: This is an abstract class that cannot be used directly.

**add_initial_equilibrium** *(initial_equilibrium: tc_python.step_or_map_diagrams.InitialEquilibrium)*
Add initial equilibrium start points from which a phase diagram is calculated.

Scans along the axis variables and generates start points when the scan procedure crosses a phase boundary.

It may take a little longer to execute than using the minimum number of start points, as some lines may be calculated more than once. But the core remembers all node points and subsequently stops calculations along a line when it finds a known node point.

It is also possible to create a sequence of start points from one initial equilibria.

- **Parameters** **initial_equilibrium** – The initial equilibrium
- **Returns** This *PhaseDiagramCalculation* object

**abstract calculate** *(keep_previous_results: bool = False, timeout_in_minutes: float = 0.0) → tc_python.step_or_map_diagrams.PhaseDiagramResult*
disable_global_minimization()
Disables global minimization.

**Default:** Enabled

**Returns** This PhaseDiagramCalculation object

dont_keep_default_equilibria()
Do not keep the initial equilibria added by default.

This is only relevant in combination with add_initial_equilibrium().

This is the default behavior.

**Returns** This PhaseDiagramCalculation object

enable_global_minimization()
Enables global minimization.

**Default:** Enabled

**Returns** This PhaseDiagramCalculation object

get_components() → List[str]
Returns the names of the components in the system (including all components auto-selected by the database(s)).

**Returns** The component names

get_gibbs_energy_addition_for(phase: str) → float
Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters** phase – Specify the name of the (stoichiometric or solution) phase with the addition

**Returns** Gibbs energy addition to G per mole formula unit.

get_system_data() → tc_python.abstract_base.SystemData
Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using with_system_modifications().

**Note:** Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.

**Returns** The system data

keep_default_equilibria()
Keep the initial equilibria added by default. This is only relevant in combination with add_initial_equilibrium().

Default behavior is to not keep default equilibria.

**Returns** This PhaseDiagramCalculation object

remove_all_initial_equilibria()
Removes all previously added initial equilibria.

**Returns** This PhaseDiagramCalculation object
run_poly_command(command: str)
    Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

    Parameters  command – The Thermo-Calc Console Mode command
    Returns This PhaseDiagramCalculation object

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

set_gibbs_energy_addition_for(phase: str, gibbs_energy: float)
    Used to specify the additional energy term (always being a constant) of a given phase. The value (gibbs_energy) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

    It is not composition-, temperature- or pressure-dependent.

    Parameters
        • phase – Specify the name of the (stoichiometric or solution) phase with the addition
        • gibbs_energy – Addition to G per mole formula unit
    Returns This PhaseDiagramCalculation object

set_phase_to_dormant(phase: str)
    Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

    Parameters phase – The phase name or ALL_PHASES for all phases
    Returns This PhaseDiagramCalculation object

set_phase_to_entered(phase: str, amount: float = 1.0)
    Sets the phase to the status ENTERED, that is the default state.

    Parameters
        • phase – The phase name or ALL_PHASES for all phases
        • amount – The phase fraction (between 0.0 and 1.0)
    Returns This PhaseDiagramCalculation object

set_phase_to_fixed(phase: str, amount: float)
    Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

    Parameters
        • phase – The phase name
        • amount – The fixed phase fraction (between 0.0 and 1.0)
    Returns This PhaseDiagramCalculation object

set_phase_to_suspended(phase: str)
    Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.
Parameters **phase** – The phase name or \texttt{ALL\_PHASES} for all phases

Returns This \texttt{PhaseDiagramCalculation} object

**with\_options** (*options: tc\_python.step\_or\_map\_diagrams.PhaseDiagramOptions*)

Sets the simulation options.

Parameters **options** – The simulation options

Returns This \texttt{PhaseDiagramCalculation} object

**with\_reference\_state** (*component: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0*)

The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

Parameters

\begin{itemize}
  \item \texttt{component} – The name of the element must be given.
  \item \texttt{phase} – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
  \item \texttt{temperature} – The Temperature (in K) for the reference state. Or \texttt{CURRENT\_TEMPERATURE} which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
  \item \texttt{pressure} – The Pressure (in Pa) for the reference state.
\end{itemize}

Returns This \texttt{PhaseDiagramCalculation} object

**with\_system\_modifications** (*system\_modifications: tc\_python.abstract\_base.SystemModifications*)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

| Note: | This is only possible if the system has been read from unencrypted (i.e. \texttt{user}) databases loaded as a \texttt{*.tdb}-file. |

Parameters **system\_modifications** – The system modification to be performed

Returns This \texttt{PhaseDiagramCalculation} object
class tc_python.step_or_map_diagrams.AbstractPropertyDiagramCalculation(calculator)
Bases: tc_python.abstract_base.AbstractCalculation

Abstract configuration required for a property diagram calculation.

**Note:** This is an abstract class that cannot be used directly.

**abstract** `calculate`(*keep_previous_results: bool = False, timeout_in_minutes: float = 0.0)*

Disables global minimization.

**Default:** Enabled

**Returns** This `PropertyDiagramCalculation` object

**disable_global_minimization**()

Disables step separate phases. This is the default setting.

**Returns** This `PropertyDiagramCalculation` object

**enable_global_minimization**()

Enables global minimization.

**Default:** Enabled

**Returns** This `PropertyDiagramCalculation` object

**enable_step_separate_phases**()

Enables step separate phases.

**Default:** By default separate phase stepping is disabled

**Note:** This is an advanced option, it is used mostly to calculate how the Gibbs energy for a number of phases varies for different compositions. This is particularly useful to calculate Gibbs energies for complex phases with miscibility gaps and for an ordered phase that is never disordered (e.g. SIGMA-phase, G-phase, MU-phase, etc.).

**Returns** This `PropertyDiagramCalculation` object

**get_components**() → List[str]

Returns the names of the components in the system (including all components auto-selected by the database(s)).

**Returns** The component names

**get_gibbs_energy_addition_for**(*phase: str*) → float

Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters** `phase` – Specify the name of the (stoichiometric or solution) phase with the addition

**Returns** Gibbs energy addition to G per mole formula unit.
get_system_data() → tc_python.abstract_base.SystemData
Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using with_system_modifications().

Note: Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.

Returns The system data

run_poly_command(command: str)
Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Parameters command – The Thermo-Calc Console Mode command

Returns This PropertyDiagramCalculation object

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

set_gibbs_energy_addition_for(phase: str, gibbs_energy: float)
Used to specify the additional energy term (always being a constant) of a given phase. The value (gibbs_energy) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

Parameters

• phase – Specify the name of the (stoichiometric or solution) phase with the addition

• gibbs_energy – Addition to G per mole formula unit

Returns This PropertyDiagramCalculation object

set_phase_to_dormant(phase: str)
Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

Parameters phase – The phase name or ALL_PHASES for all phases

Returns This PropertyDiagramCalculation object

set_phase_to_entered(phase: str, amount: float = 1.0)
Sets the phase to the status ENTERED, that is the default state.

Parameters

• phase – The phase name or ALL_PHASES for all phases

• amount – The phase fraction (between 0.0 and 1.0)

Returns This PropertyDiagramCalculation object

set_phase_to_fixed(phase: str, amount: float)
Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.
Parameters

- **phase** – The phase name
- **amount** – The fixed phase fraction (between 0.0 and 1.0)

Returns This PropertyDiagramCalculation object

**set_phase_to_suspended**(phase: str)
Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

Parameters **phase** – The phase name or ALL_PHASES for all phases

Returns This PropertyDiagramCalculation object

**with_options**(options: tc_python.step_or_map_diagrams.PropertyDiagramOptions)
Sets the simulation options.

Parameters **options** – The simulation options

Returns This PropertyDiagramCalculation object

**with_reference_state**(component: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0)
The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

Parameters

- **component** – The name of the element must be given.
- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
- **pressure** – The Pressure (in Pa) for the reference state.

Returns This PropertyDiagramCalculation object

**with_system_modifications**(system_modifications: tc_python.abstract_base.SystemModifications)
Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).
Note: This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a *.tdb-file.

Parameters system_modifications – The system modification to be performed

Returns This PropertyDiagramCalculation object

class tc_python.step_or_map_diagrams.AxisType
    Bases: tc_python.step_or_map_diagrams.AbstractAxisType

Factory class providing objects for configuring a logarithmic or linear axis by using AxisType.linear() or AxisType.logarithmic().

classmethod linear()
    Creates an object for configuring a linear calculation axis.

    Default: A minimum number of 40 steps.

    Note: The returned object can be configured regarding the maximum step size or the minimum number of steps on the axis.

    Returns A new Linear object

classmethod logarithmic()
    Creates an object for configuring a logarithmic calculation axis.

    Default: A scale factor of 1.1

    Note: The returned object can be configured regarding the scale factor.

    Returns A new Logarithmic object

class tc_python.step_or_map_diagrams.CalculationAxis(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str])
    Bases: object

    A calculation axis used for property and phase diagram calculations.

    Note: A calculation axis is defining the varied condition and the range of variation. It is the same concept as in Thermo-Calc Graphical Mode or Console Mode.

    Default: A Linear axis with a minimum number of 40 steps

set_max (max: float)
    Sets the maximum quantity value of the calculation axis.

    There is no default value set, it always needs to be defined.

    Parameters max – The maximum quantity value of the axis [unit according to the axis quantity]

    Returns This CalculationAxis object
set_min(min: float)
Sets the minimum quantity value of the calculation axis.

There is no default value set, it always needs to be defined.

Parameters min – The minimum quantity value of the axis [unit according to the axis quantity]

Returns This CalculationAxis object

set_start_at(at: float)
Sets the starting point of the calculation on the axis.

Default: The default starting point is the center between the minimum and maximum quantity value

Parameters at – The starting point on the axis [unit according to the axis quantity]

Returns This CalculationAxis object

with_axis_type(axis_type: tc_python.step_or_map_diagrams.AxisType)
Sets the type of the axis.

Default: A Linear axis with a minimum number of 40 steps

Parameters axis_type – The axis type (linear or logarithmic)

Returns This CalculationAxis object

class tc_python.step_or_map_diagrams.Direction(value)
Bases: enum.Enum
An enumeration.

DECREASE_FIRST_AXIS = 3
DECREASE_SECOND_AXIS = 4
INCREASE_FIRST_AXIS = 0
INCREASE_SECOND_AXIS = 1

class tc_python.step_or_map_diagrams.InitialEquilibrium(first_axis: float, second_axis: float)
Bases: object

add_equilibria_at_all_phase_changes()
This generates one start point for each set of phase change in the chosen direction of the specified axis
This ensures finding all possible phase boundary lines (not just the first one) along such an axis direction.

Default behavior is to only generate one start point at the first phase change.

Returns This InitialEquilibrium object

add_equilibria_at_first_phase_change()
This generates one start point at the first phase change.
This is the default behavior.

Returns This InitialEquilibrium object

set_direction(direction_enum: tc_python.step_or_map_diagrams.Direction)
Specifies along which axes the initial equilibria should be added.
The default direction is INCREASE_FIRST_AXIS.

Parameters direction_enum –

Returns This InitialEquilibrium object

5.1. Calculations
class tc_python.step_or_map_diagrams.Linear
    Bases: tc_python.step_or_map_diagrams.AxisType

    Represents a linear axis.

    get_type() -> str
        Convenience method for getting axis type.

        Returns: The type

    set_max_step_size(max_step_size: float)
        Sets the axis to use the maximum step size configuration.

        Default: This is not the default which is minimum number of steps

        Note: Either maximum step size or minimum number of steps can be used but not both at the same time.

        Parameters max_step_size – The maximum step size [unit according to the axis quantity]

        Returns: This Linear object

    set_min_nr_of_steps(min_nr_of_steps: float = 40)
        Sets the axis to use the minimum number of steps configuration.

        Default: This is the default option (with a minimum number of steps of 40)

        Note: Either maximum step size or minimum number of steps can be used but not both at the same time.

        Parameters min_nr_of_steps – The minimum number of steps

        Returns: This Linear object

class tc_python.step_or_map_diagrams.Logarithmic(scale_factor: float = 1.1)
    Bases: tc_python.step_or_map_diagrams.AxisType

    Represents a logarithmic axis.

    get_type() -> str
        Convenience method for getting axis type.

        Returns: The type

    set_scale_factor(scale_factor: float = 1.1)
        Sets the scale factor.

        Default: 1.1

        Parameters scale_factor – The scale factor setting the maximum factor between two calculated values, must be larger than 1.0

        Returns: This Logarithmic object
class tc_python.step_or_map_diagrams.PhaseDiagramCalculation(calculator)
Bases: tc_python.step_or_map_diagrams.AbstractPhaseDiagramCalculation

Configuration for a phase diagram calculation.

Note: Specify the conditions, the calculation is performed with `calculate()`.

add_initial_equilibrium(initial_equilibrium: tc_python.step_or_map_diagrams.InitialEquilibrium)
Add initial equilibrium start points from which a phase diagram is calculated.
Scans along the axis variables and generates start points when the scan procedure crosses a phase boundary.
It may take a little longer to execute than using the minimum number of start points, as some lines may be calculated more than once. But the core remembers all node points and subsequently stops calculations along a line when it finds a known node point.
It is also possible to create a sequence of start points from one initial equilibria.

Parameters

initial_equilibrium – The initial equilibrium

Returns This PhaseDiagramCalculation object

calculate(keep_previous_results: bool = False, timeout_in_minutes: float = 0.0) →
tc_python.step_or_map_diagrams.PhaseDiagramResult
Performs the phase diagram calculation.

Warning: If you use `keep_previous_results=True`, you must not use another calculator or even get results in between the calculations using `calculate()`. Then the previous results will actually be lost.

Parameters

• keep_previous_results – If True, results from any previous call to this method are appended. This can be used to combine calculations with multiple start points if the mapping fails at a certain condition.

• timeout_in_minutes – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than `timeout_in_minutes`, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

Returns A new PhaseDiagramResult object which later can be used to get specific values from the calculated result.

disable_global_minimization()
Disables global minimization.

Default: Enabled

Returns This PhaseDiagramCalculation object

dont_keep_default_equilibria()
Do not keep the initial equilibria added by default.
This is only relevant in combination with `add_initial_equilibrium()`.
This is the default behavior.

Returns This PhaseDiagramCalculation object
enable_global_minimization()

Enables global minimization.

**Default**: Enabled

**Returns** This `PhaseDiagramCalculation` object

get_components() \(\rightarrow\) List[str]

Returns the names of the components in the system (including all components auto-selected by the database(s)).

**Returns** The component names

get_gibbs_energy_addition_for(phase: str) \(\rightarrow\) float

Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters** phase – Specify the name of the (stoichiometric or solution) phase with the addition

**Returns** Gibbs energy addition to G per mole formula unit.

get_system_data() \(\rightarrow\) tc_python.abstract_base.SystemData

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

**Note:** Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.

**Returns** The system data

keep_default_equilibria()

Keep the initial equilibria added by default. This is only relevant in combination with `add_initial_equilibrium()`.

Default behavior is to not keep default equilibria.

**Returns** This `PhaseDiagramCalculation` object

remove_all_conditions()

Removes all set conditions.

**Returns** This `PhaseDiagramCalculation` object

remove_all_initial_equilibria()

Removes all previously added initial equilibria.

**Returns** This `PhaseDiagramCalculation` object

remove_condition(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str])

Removes the specified condition.

**Parameters** quantity – The thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example `X(Cr)`)

**Returns** This `ThermodynamicCalculation` object

run_poly_command(command: str)

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.
Parameters **command** – The Thermo-Calc Console Mode command

**Returns** This *PhaseDiagramCalculation* object

**Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

**Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

```python
set_condition(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], value: float)
```

Sets the specified condition.

**Parameters**

- **quantity** – The thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example X(Cr))

- **value** – The value of the condition

**Returns** This *PhaseDiagramCalculation* object

```python
set_gibbs_energy_addition_for(phase: str, gibbs_energy: float)
```

Used to specify the additional energy term (always being a constant) of a given phase. The value (`gibbs_energy`) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters**

- **phase** – Specify the name of the (stoichiometric or solution) phase with the addition

- **gibbs_energy** – Addition to G per mole formula unit

**Returns** This *PhaseDiagramCalculation* object

```python
set_phase_to_dormant(phase: str)
```

Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

**Parameters** **phase** – The phase name or `ALL_PHASES` for all phases

**Returns** This *PhaseDiagramCalculation* object

```python
set_phase_to_entered(phase: str, amount: float = 1.0)
```

Sets the phase to the status ENTERED, that is the default state.

**Parameters**

- **phase** – The phase name or `ALL_PHASES` for all phases

- **amount** – The phase fraction (between 0.0 and 1.0)

**Returns** This *PhaseDiagramCalculation* object

```python
set_phase_to_fixed(phase: str, amount: float)
```

Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

**Parameters**

---

5.1. Calculations
• **phase** – The phase name

• **amount** – The fixed phase fraction (between 0.0 and 1.0)

**Returns** This PhaseDiagramCalculation object

**set_phase_to_suspended** *(phase: str)*

Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

**Parameters** phase – The phase name or ALL_PHASES for all phases

**Returns** This PhaseDiagramCalculation object

**with_first_axis** *(axis: tc_python.step_or_map_diagrams.CalculationAxis)*

Sets the first calculation axis.

**Parameters** axis – The axis

**Returns** This PhaseDiagramCalculation object

**with_options** *(options: tc_python.step_or_map_diagrams.PhaseDiagramOptions)*

Sets the simulation options.

**Parameters** options – The simulation options

**Returns** This PhaseDiagramCalculation object

**with_reference_state** *(component: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0)*

The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

**Parameters**

- **component** – The name of the element must be given.

- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.

- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.

- **pressure** – The Pressure (in Pa) for the reference state.

**Returns** This PhaseDiagramCalculation object
with_second_axis

Sets the second calculation axis.

Parameters

axis – The axis

Returns This PhaseDiagramCalculation object

with_system_modifications

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Parameters

system_modifications – The system modification to be performed

Returns This PhaseDiagramCalculation object

class tc_python.step_or_map_diagrams.PhaseDiagramOptions

Bases: object

Simulation options for phase diagram calculations.

disable_approximate_driving_force_for_metastable_phases()

Disables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

If it is important that these driving forces are correct, use disable_approximate_driving_force_for_metastable_phases() to force the calculation to converge for the metastable phases.

Returns This PhaseDiagramOptions object

disable_control_step_size_during_minimization()

Disables stepsize control during minimization (non-global).

Default: Enabled

Returns This PhaseDiagramOptions object

disable_force_positive_definite_phase_hessian()

Disables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for “Hessian minimization”.

Default: Enabled

Returns This PhaseDiagramOptions object

dont_use_auto_start_points()

Switches the usage of automatic starting points for the mapping off.

Default: Switched on

Returns This PhaseDiagramOptions object
**dont_use_inside_mesh_points()**

Switches the usage of inside meshing points for the mapping off.

Default: Switched off

Returns This `PhaseDiagramOptions` object

**enable_approximate_driving_force_for_metastable_phases()**

Enables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

If it is important that these driving forces are correct, use `disable_approximate_driving_force_for_metastable_phases()` to force the calculation to converge for the metastable phases.

Returns This `PhaseDiagramOptions` object

**enable_control_step_size_during_minimization()**

Enables stepsize control during normal minimization (non-global).

Default: Enabled

Returns This `PhaseDiagramOptions` object

**enable_force_positive_definite_phase_hessian()**

Enables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for “Hessian minimization”.

Default: Enabled

Returns This `PhaseDiagramOptions` object

**set_global_minimization_max_grid_points** *(max_grid_points: int = 2000)*

Sets the maximum number of grid points in global minimization. **Only applicable if global minimization is actually used**.

Default: 2000 points

Parameters `max_grid_points` – The maximum number of grid points

Returns This `PhaseDiagramOptions` object

**set_global_minimization_test_interval** *(global_test_interval: int = 0)*

Sets the interval for the global test.

Default: 0

Parameters `global_test_interval` – The global test interval

Returns This `PhaseDiagramOptions` object

**set_max_no_of_iterations** *(max_no_of_iterations: int = 500)*

Set the maximum number of iterations.

Default: max. 500 iterations
Note: As some models give computation times of more than 1 CPU second/iteration, this number is also used to check the CPU time and the calculation stops if 500 CPU seconds/iterations are used.

**Parameters** `max_no_of_iterations` – The max. number of iterations

**Returns** This `PhaseDiagramOptions` object

`set_no_of_mesh_along_axis(no_of_mesh_along_axis: int = 3)`
Sets the number of meshes along an axis for the mapping.

Default: 3

**Parameters** `no_of_mesh_along_axis` – The number of meshes

**Returns** This `PhaseDiagramOptions` object

`set_required_accuracy(accuracy: float = 1e-06)`
Sets the required relative accuracy.

Default: 1.0E-6

Note: This is a relative accuracy, and the program requires that the relative difference in each variable must be lower than this value before it has converged. A larger value normally means fewer iterations but less accurate solutions. The value should be at least one order of magnitude larger than the machine precision.

**Parameters** `accuracy` – The required relative accuracy

**Returns** This `PhaseDiagramOptions` object

`set_smallest_fraction(smallest_fraction: float = 1e-12)`
Sets the smallest fraction for constituents that are unstable.

It is normally only in the gas phase that you can find such low fractions.

The default value for the smallest site-fractions is 1E-12 for all phases except for IDEAL phase with one sublattice site (such as the GAS mixture phase in many databases) for which the default value is always as 1E-30.

**Parameters** `smallest_fraction` – The smallest fraction for constituents that are unstable

**Returns** This `PhaseDiagramOptions` object

`use_auto_start_points()`
Switches the usage of automatic starting points for the mapping on.

Default: Switched on

**Returns** This `PhaseDiagramOptions` object

`use_inside_mesh_points()`
Switches the usage of inside meshing points for the mapping off.

Default: Switched off

**Returns** This `PhaseDiagramOptions` object
class tc_python.step_or_map_diagrams.PhaseDiagramResult(result)
Bases: tc_python.abstract_base.AbstractResult

Result of a phase diagram calculation, it can be evaluated using quantities or Console Mode syntax.

add_coordinate_for_phase_label (x: float, y: float)
Sets a coordinate in the result plot for which the stable phases will be evaluated and provided in the
result data object. This can be used to plot the phases of a region into the phase diagram or just to
programmatically evaluate the phases in certain regions.

**Warning:** This method takes coordinates of the **plot** axes and not of the calculation axis.

**Parameters**

- x – The coordinate of the first **plot** axis (“x-axis”) [unit of the **plot** axis]
- y – The coordinate of the second **plot** axis (“y-axis”) [unit of the **plot** axis]

**Returns** This PhaseDiagramResult object

get_values_grouped_by_quantity_of (x_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], y_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str]) → tc_python.step_or_map_diagrams.PhaseDiagramResultValues

Returns x-y-line data grouped by the multiple datasets of the specified quantities (for example in depen-
dency of components). The available quantities can be found in the documentation of the factory class
ThermodynamicQuantity. Usually the result data represents the phase diagram.

**Note:** The different datasets will contain NaN-values between different subsections and are not sorted
(because they are unsortable due to their nature).

**Note:** Its possible to use functions as axis variables, either by using ThermodynamicQuan-
tity.user_defined_function, or by using an expression that contains ‘=’.

**Example**

get_values_grouped_by_quantity_of('T', ThermodynamicQuan-
ty.user_defined_function('HM.T'))

**Example**

get_values_grouped_by_quantity_of('T', 'CP=HM.T')

**Parameters**

- x_quantity – The first quantity (“x-axis”), Console Mode syntax strings can be used
  as an alternative (for example ‘T’), or even a function (for example ‘f=T*1.01’)
- y_quantity – The second quantity (“y-axis”), Console Mode syntax strings can be used
  as an alternative (for example ‘NV’), or even a function (for example ‘CP=HM.T’)

**Returns** The phase diagram data

get_values_grouped_by_stable_phases_of (x_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], y_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str]) → tc_python.step_or_map_diagrams.PhaseDiagramResultValues

Returns x-y-line data grouped by the sets of “stable phases” (for example “LIQUID” or “LIQUID
+ FCC_A1”). The available quantities can be found in the documentation of the factory class
ThermodynamicQuantity. Usually the result data represents the phase diagram.
**Note:** The different datasets will contain *NaN*-values between different subsections and are not sorted (because they are unsortable due to their nature).

**Note:** Its possible to use functions as axis variables, either by using *ThermodynamicQuantity.user_defined_function*, or by using an expression that contains ‘=’.

**Example**

```python
get_values_grouped_by_quantity_of('T', ThermodynamicQuantity.user_defined_function('HM.T'))
```

**Example**

```python
get_values_grouped_by_quantity_of('T', 'CP=HM.T')
```

**Parameters**

- **x_quantity** – The first quantity (“x-axis”), Console Mode syntax strings can be used as an alternative (for example ‘T’), or even a function (for example ‘f=T*1.01’)

- **y_quantity** – The second quantity (“y-axis”), Console Mode syntax strings can be used as an alternative (for example ‘NV’), or even a function (for example ‘CP=HM.T’)

**Returns**

The phase diagram data

`remove_phase_labels()`

Erases all added coordinates for phase labels.

**Returns**

This *PhaseDiagramResult* object

`save_to_disk(path: str)`

Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with *load_result_from_disk()*

**Parameters**

- **path** – the path to the folder you want the result to be saved in. It can be relative or absolute.

**Returns**

this *PhaseDiagramResult* object

`set_phase_name_style(phase_name_style_enum: tc_python.step_or_map_diagrams.PhaseNameStyle = <PhaseNameStyle.NONE: 0>)`

Sets the style of the phase name labels that will be used in the result data object (constitution description, ordering description, ...).

**Default:** PhaseNameStyle.NONE

**Parameters**

- **phase_name_style_enum** – The phase name style

**Returns**

This *PhaseDiagramResult* object

---

**Note:** The datasets will normally contain different sections separated by *NaN*-values.

**Returns**

The invariants dataset object
get_lines() → Dict[str, tc_python.utils.ResultValueGroup]
Returns the x- and y-datasets of all phase boundaries in the phase diagram.

Note: The datasets will normally contain different sections separated by NaN-values.

Returns Containing the phase boundary datasets with the quantities or stable phases as keys (depending on the used method to get the values)

get_phase_labels() → List[tc_python.step_or_map_diagrams.PhaseLabel]
Returns the phase labels added for certain coordinates using PhaseDiagramResult.add_coordinate_for_phase_label().

Returns The list with the phase label data (that contains plot coordinates and stable phases)

get_tie_lines() → tc_python.utils.ResultValueGroup
Returns the x- and y-datasets of all tie-lines in the phase diagram.

Note: The datasets will normally contain different sections separated by NaN-values.

Returns The tie-line dataset object

class tc_python.step_or_map_diagrams.PhaseLabel(phase_label_java)
Bases: object
Represents a phase label at a plot coordinate, i.e. the stable phases that are present at that plot coordinate.

get_text() → str
Accessor for the phase label :return: the phase label

get_x() → List[float]
Accessor for the x-value :return: the x value

get_y() → List[float]
Accessor for the y-value :return: the y value

class tc_python.step_or_map_diagrams.PhaseNameStyle(value)
Bases: enum.Enum
The style of the phase names used in the labels.

ALL = 1
Adding ordering and constitution description.

CONSTITUTION_DESCRIPTION = 3
Adding only constitution description.

NONE = 0
Only the phase names.

ORDERING_DESCRIPTION = 4
Adding only ordering description.

class tc_python.step_or_map_diagrams.PropertyDiagramCalculation(calculator)
Bases: tc_python.step_or_map_diagrams.AbstractPropertyDiagramCalculation

calculate(keep_previous_results: bool = False, timeout_in_minutes: float = 0.0) →
  tc_python.step_or_map_diagrams.PropertyDiagramResult
Performs the property diagram calculation.
Warning: If you use `keep_previous_results=True`, you must not use another calculator or even get results in between the calculations using `calculate()`. Then the previous results will actually be lost.

Parameters

- **keep_previous_results** – If True, results from any previous call to this method are appended. This can be used to combine calculations with multiple start points if the stepping fails at a certain condition.

- **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

Returns A new PropertyDiagramResult object which later can be used to get specific values from the calculated result

disable_global_minimization()
Disables global minimization.

Default: Enabled

Returns This PropertyDiagramCalculation object

disable_step_separate_phases()
Disables step separate phases. This is the default setting.

Returns This PropertyDiagramCalculation object

enable_global_minimization()
Enables global minimization.

Default: Enabled

Returns This PropertyDiagramCalculation object

enable_step_separate_phases()
Enables step separate phases.

Default: By default separate phase stepping is disabled

Note: This is an advanced option, it is used mostly to calculate how the Gibbs energy for a number of phases varies for different compositions. This is particularly useful to calculate Gibbs energies for complex phases with miscibility gaps and for an ordered phase that is never disordered (e.g. SIGMA-phase, G-phase, MU-phase, etc.).

Returns This PropertyDiagramCalculation object

get_components() → List[str]
Returns the names of the components in the system (including all components auto-selected by the database(s)).

Returns The component names

get_gibbs_energy_addition_for(phase: str) → float
Used to get the additional energy term (always being a constant) of a given phase. The value given is added
to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface
tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters**

- **phase** – Specify the name of the (stoichiometric or solution) phase with the addition

**Returns**

Gibbs energy addition to $G$ per mole formula unit.

**get_system_data()** → `tc_python.abstract_base.SystemData`

Returns the content of the database for the currently loaded system. This can be used to modify the param-
eters and functions and to change the current system by using `with_system_modifications()`.

**Note:** Parameters can only be read from unencrypted (i.e. `user`) databases loaded as `*.tdb`-file.

**Returns**

The system data

**remove_all_conditions()**

Removes all set conditions.

**Returns**

This `PropertyDiagramCalculation` object

**remove_condition(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str])**

Removes the specified condition.

**Parameters**

- **quantity** – The thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example $X(Cr)$)

**Returns**

This `PropertyDiagramCalculation` object

**run_poly_command(command: str)**

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

**Parameters**

- **command** – The Thermo-Calc Console Mode command

**Returns**

This `PropertyDiagramCalculation` object

**Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

**Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

**set_condition(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], value: float)**

Sets the specified condition.

**Parameters**

- **quantity** – The thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example $X(Cr)$)

- **value** – The value of the condition

**Returns**

This `PropertyDiagramCalculation` object
**set_gibbs_energy_addition_for** *(phase: str, gibbs_energy: float)*

Used to specify the additional energy term (always being a constant) of a given phase. The value *(gibbs_energy)* given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters**

- **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
- **gibbs_energy** – Addition to G per mole formula unit

**Returns** This *PropertyDiagramCalculation* object

**set_phase_to_dormant** *(phase: str)*

Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

**Parameters** **phase** – The phase name or *ALL_PHASES* for all phases

**Returns** This *PropertyDiagramCalculation* object

**set_phase_to_entered** *(phase: str, amount: float = 1.0)*

Sets the phase to the status ENTERED, that is the default state.

**Parameters**

- **phase** – The phase name or *ALL_PHASES* for all phases
- **amount** – The phase fraction (between 0.0 and 1.0)

**Returns** This *PropertyDiagramCalculation* object

**set_phase_to_fixed** *(phase: str, amount: float)*

Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

**Parameters**

- **phase** – The phase name
- **amount** – The fixed phase fraction (between 0.0 and 1.0)

**Returns** This *PropertyDiagramCalculation* object

**set_phase_to_suspended** *(phase: str)*

Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

**Parameters** **phase** – The phase name or *ALL_PHASES* for all phases

**Returns** This *PropertyDiagramCalculation* object

**with_axis** *(axis: tc_python.step_or_map_diagrams.CalculationAxis)*

Sets the calculation axis.

**Parameters** **axis** – The axis

**Returns** This *PropertyDiagramCalculation* object

**with_options** *(options: tc_python.step_or_map_diagrams.PropertyDiagramOptions)*

Sets the simulation options.

**Parameters** **options** – The simulation options

**Returns** This *PropertyDiagramCalculation* object
with_reference_state (component: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0)

The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

Parameters

- **component** – The name of the element must be given.
- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
- **pressure** – The Pressure (in Pa) for the reference state.

Returns This PropertyDiagramCalculation object

with_system_modifications (system_modifications: tc_python.abstract_base.SystemModifications)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a *.tdb-file.

Parameters **system_modifications** – The system modification to be performed

Returns This PropertyDiagramCalculation object

class tc_python.step_or_map_diagrams.PropertyDiagramOptions

Bases: object

Simulation options for the property diagram calculations.

disable_approximate_driving_force_for_metastable_phases ()

Disables the approximation of the driving force for metastable phases.

Default: Enabled
Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

If it is important that these driving forces are correct, use `disable_approximate_driving_force_for_metastable_phases()` to force the calculation to converge for the metastable phases.

Returns This `PropertyDiagramOptions` object

`disable_control_step_size_during_minimization()` Disables stepsize control during minimization (non-global).

Default: Enabled

Returns This `PropertyDiagramOptions` object

`disable_force_positive_definite_phase_hessian()` Disables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for “Hessian minimization”.

Default: Enabled

Returns This `PropertyDiagramOptions` object

`enable_approximate_driving_force_for_metastable_phases()` Enables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

If it is important that these driving forces are correct, use `disable_approximate_driving_force_for_metastable_phases()` to force the calculation to converge for the metastable phases.

Returns This `PropertyDiagramOptions` object

`enable_control_step_size_during_minimization()` Enables stepsize control during normal minimization (non-global).

Default: Enabled

Returns This `PropertyDiagramOptions` object

`enable_force_positive_definite_phase_hessian()` Enables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for “Hessian minimization”.

Default: Enabled

Returns This `PropertyDiagramOptions` object

`set_global_minimization_max_grid_points` *(max_grid_points: int = 2000)* Sets the maximum number of grid points in global minimization. Only applicable if global minimization is actually used.
Default: 2000 points

Parameters **max_grid_points** – The maximum number of grid points

Returns This `PropertyDiagramOptions` object

**set_global_minimization_test_interval** (`global_test_interval: int = 0`)  
Sets the interval for the global test.

Default: 0

Parameters **global_test_interval** – The global test interval

Returns This `PropertyDiagramOptions` object

**set_max_no_of_iterations** (`max_no_of_iterations: int = 500`)  
Set the maximum number of iterations.

Default: max. 500 iterations

**Note:** As some models give computation times of more than 1 CPU second/iteration, this number is also used to check the CPU time and the calculation stops if 500 CPU seconds/iterations are used.

Parameters **max_no_of_iterations** – The max. number of iterations

Returns This `PropertyDiagramOptions` object

**set_required_accuracy** (`accuracy: float = 1e-06`)  
Sets the required relative accuracy.

Default: 1.0E-6

**Note:** This is a relative accuracy, and the program requires that the relative difference in each variable must be lower than this value before it has converged. A larger value normally means fewer iterations but less accurate solutions. The value should be at least one order of magnitude larger than the machine precision.

Parameters **accuracy** – The required relative accuracy

Returns This `PropertyDiagramOptions` object

**set_smallest_fraction** (`smallest_fraction: float = 1e-12`)  
Sets the smallest fraction for constituents that are unstable.

It is normally only in the gas phase that you can find such low fractions.

The **default value** for the smallest site-fractions is 1E-12 for all phases except for IDEAL phase with one sublattice site (such as the GAS mixture phase in many databases) for which the default value is always as 1E-30.

Parameters **smallest_fraction** – The smallest fraction for constituents that are unstable

Returns This `PropertyDiagramOptions` object

**class** `tc_python.step_or_map_diagrams.PropertyDiagramResult` (*result*)  

**Bases:** `tc_python.abstract_base.AbstractResult`  

Result of a property diagram. This can be used to query for specific values.
get_values_grouped_by_quantity_of(x_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], y_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], sort_and_merge: bool = True) \rightarrow Dict[str, tc_python.utils.ResultValueGroup]

Returns x-y-line data grouped by the multiple datasets of the specified quantities (typically the phases). The available quantities can be found in the documentation of the factory class ThermodynamicQuantity.

**Note:** The different datasets might contain NaN-values between different subsections and might not be sorted even if the flag `sort_and_merge` has been set (because they might be unsortable due to their nature).

**Parameters**

- **x_quantity** – The first quantity (“x-axis”), Console Mode syntax strings can be used as an alternative (for example ‘T’), or even a function (for example ‘f=T*1.01’)
- **y_quantity** – The second quantity (“y-axis”), Console Mode syntax strings can be used as an alternative (for example ‘NV’), or even a function (for example ‘CP=HM.T’)
- **sort_and_merge** – If True, the data is sorted and merged into as few subsections as possible (divided by NaN)

**Returns** Containing the datasets with the quantities as their keys

get_values_grouped_by_stable_phases_of(x_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], y_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], sort_and_merge: bool = True) \rightarrow Dict[str, tc_python.utils.ResultValueGroup]

Returns x-y-line data grouped by the sets of “stable phases” (for example “LIQUID” or “LIQUID + FCC_A1”). The available quantities can be found in the documentation of the factory class ThermodynamicQuantity.

**Note:** The different datasets might contain NaN-values between different subsections and different lines of an ambiguous dataset. They might not be sorted even if the flag `sort_and_merge` has been set (because they might be unsortable due to their nature).

**Parameters**

- **x_quantity** – The first quantity (“x-axis”), Console Mode syntax strings can be used as an alternative (for example ‘T’), or even a function (for example ‘f=T*1.01’)
- **y_quantity** – The second quantity (“y-axis”), Console Mode syntax strings can be used as an alternative (for example ‘NV’), or even a function (for example ‘CP=HM.T’)
- **sort_and_merge** – If True, the data is sorted and merged into as few subsections as possible (divided by NaN)

**Returns** Containing the datasets with the quantities as their keys
Parameters

- **x_quantity** – The first quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example 'T'), or even a function (for example 'f=T*1.01')

- **y_quantity** – The second quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example 'NV'), or even a function (for example 'CP=HM.T')

- **sort_and_merge** – If True, the data will be sorted and merged into as few subsections as possible (divided by NaN)

Returns
Containing the datasets with the quantities as their keys

```
get_values_of(x_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str],
              y_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str]) → [typing.List[float], typing.List[float]]
```

Returns sorted x-y-line data without any separation. Use `get_values_grouped_by_quantity_of()` or `get_values_grouped_by_stable_phases_of()` instead if you need such a separation. The available quantities can be found in the documentation of the factory class ThermodynamicQuantity.

**Note:** This method will always return sorted data without any NaN-values. If it is unsortable that might give data that is hard to interpret. In such a case you need to choose the quantity in another way or use one of the other methods. One example of this is to use quantities with All-markers, for example MassFractionOfAComponent("All").

**Note:** It's possible to use functions as axis variables, either by using ThermodynamicQuantity.user_defined_function, or by using an expression that contains '='.

**Example**
```
get_values_grouped_by_quantity_of('T', ThermodynamicQuantity.user_defined_function('HM.T'))
```

**Example**
```
get_values_grouped_by_quantity_of('T', 'CP=HM.T')
```

Parameters

- **x_quantity** – The first Thermodynamic quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example 'T') or even a function (for example 'f=T*1.01')

- **y_quantity** – The second Thermodynamic quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example 'NV'), or even a function (for example 'CP=HM.T')

Returns
A tuple containing the x- and y-data in lists

```
save_to_disk(path: str)
```

Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with `load_result_from_disk()`.

**Parameters**

- **path** – the path to the folder you want the result to be saved in. It can be relative or absolute.

**Returns** this PropertyDiagramResult object
**set_phase_name_style** (phase_name_style_enum: tc_python.step_or_map_diagrams.PhaseNameStyle = <PhaseNameStyle.NONE: 0>)

Sets the style of the phase name labels that will be used in the result data object (constitution description, ordering description, ...).

**Default:** PhaseNameStyle.NONE

**Parameters**

- **phase_name_style_enum** – The phase name style

**Returns**

This PropertyDiagramResult object

### 5.1.6 Module “diffusion”

**class** tc_python.diffusion.AbstractBoundaryCondition  
Bases: object

The abstract base class for all boundary conditions.

**class** tc_python.diffusion.AbstractCalculatedGrid  
Bases: tc_python.diffusion.AbstractGrid

The abstract base class for calculated grids.

**class** tc_python.diffusion.AbstractElementProfile  
Bases: object

The abstract base class for all initial composition profile types.

**class** tc_python.diffusion.AbstractGrid  
Bases: object

The abstract base class for all grids.

**class** tc_python.diffusion.AbstractSolver  
Bases: object

Abstract base class for the solvers (Classic, Homogenization and Automatic).

**class** tc_python.diffusion.ActivityFluxFunction  
Bases: tc_python.diffusion.BoundaryCondition

**get_type** () → str

The type of the boundary condition.

**Returns**

The type

**set_flux_function** (element_name: str, f: str = '0', g: str = '1', n: float = 1.0, to_time: float = 1.7976931348623157e+308)

The flux for the independent components must be given in the format:

\[ J = f(T,P,\text{TIME}) \times (\text{ACTIVITY}^N - g(T,P,\text{TIME})) \]

where \( f \) and \( g \) may be functions of time (TIME), temperature (T), and pressure (P), and \( N \) is an integer. \( f \) and \( g \) must be expressed in DICTRA Console Mode syntax.

**Parameters**

- **element_name** – The name of the element
- **f** – the function \( f \) in the formula above
- **g** – the function \( g \) in the formula above
- **n** – the constant \( N \) in the formula above
• `to_time` – The max-time for which the flux function is used.

class tc_python.diffusion.AutomaticSolver

    Bases: tc_python.diffusion.Solver

    Solver using the homogenization model if any region has more than one phase, otherwise using the classic model.

    Note: This is the default solver and recommended for most applications.

    get_type() \rightarrow \text{str}

        The type of the solver.

        Returns The type

    set_flux_balance_equation_accuracy(accuracy: float = 1e-16)

        Only valid if the :class:`ClassicSolver` is actually used (i.e. not more than one phase in each region).

        Sets the required accuracy during the solution of the flux balance equations. Default: 1.0e-16

        Parameters accuracy – The required accuracy

        Returns A new AutomaticSolver object

    set_tieline_search_variable_to_activity()

        Only valid if the :class:`ClassicSolver` is actually used (i.e. not more than one phase in each region).

        Configures the solver to use the activity of a component to find the correct tie-line at the phase interface. Either activity or chemical potential are applied to reduce the degrees of freedom at the local equilibrium. Default: This is the default setting

        Returns A new AutomaticSolver object

    set_tieline_search_variable_to_potential()

        Only valid if the :class:`ClassicSolver` is actually used (i.e. not more than one phase in each region).

        Configures the solver to use the chemical potential of a component to find the correct tie-line at the phase interface. Either activity or chemical potential are applied to reduce the degrees of freedom at the local equilibrium. Default: To use the activity

        Returns A new AutomaticSolver object

class tc_python.diffusion.BoundaryCondition

    Bases: tc_python.diffusion.AbstractBoundaryCondition

    Contains factory methods for the the different boundary conditions available.

    classmethod activity_flux_function()

        Factory method that creates a new activity-flux-function boundary condition.

        This type of boundary condition is used to take into account the finite rate of a surface reaction.

        The flux for the independent components must be given in the format:

        \[ J = f(T,P,\text{TIME}) \cdot (\text{ACTIVITY}^N - g(T,P,\text{TIME})) \]

        where f and g may be functions of time (TIME), temperature (T), and pressure (P), and N is an integer. f and g must be expressed in DICTRA Console Mode syntax.

        Note: The activities are those with user-defined reference states. The function mass transfer coefficient
is the mass transfer coefficient, activity of the corresponding species in the gas is the activity of the corresponding species in the gas and N is a stoichiometric coefficient.

**Note:** For more details see L. Sproge and J. Ågren, “Experimental and theoretical studies of gas consumption in the gas carburizing process” J. Heat Treat. 6, 9–19 (1988).

**Returns** A new ActivityFluxFunction object

```python
classmethod closed_system()
    Factory method that creates a new closed-system boundary condition.
    Returns A new ClosedSystem object

classmethod fix_flux_value()
    Factory method that creates a new fix-flux-value boundary condition.
    This type of boundary condition makes it possible to enter functions that yield the flux times the molar volume for the independent components. May be a function of time, temperature and pressure: $J(T,P,TIME)$.
    Returns A new FixFluxValue object

classmethod fixed_compositions(unit_enum: tc_python.diffusion.Unit = <Unit.MASS_PERCENT: 3>)
    Factory method that creates a new fixed-composition boundary condition.
    Parameters unit_enum – The composition unit
    Returns A new FixedCompositions object

classmethod mixed_zero_flux_and_activity()
    Factory method that creates a new mixed zero-flux and activity boundary condition
    Returns A new MixedZeroFluxAndActivity object
```

**Returns** A new ActivityFluxFunction object

```python
class tc_python.diffusion.CalculatedGrid
    Bases: tc_python.diffusion.AbstractCalculatedGrid
    Factory class for grids generated by a mathematical series (linear, geometric, ...). Use tc_python.diffusion.PointByPointGrid instead if you want to use an existing grid from experimental data or a previous calculation.

**Note:** A region must contain a number of grid points. The composition is only known at these grid points and the software assumes that the composition varies linearly between them. The amount and composition of all the phases present at a single grid point in a certain region are those given by thermodynamic equilibrium keeping the over-all composition at the grid point fixed.

```python
classmethod double_geometric(no_of_points: int = 50, lower_geometrical_factor: float = 1.1, upper_geometrical_factor: float = 0.9)
    Factory method that creates a new double geometric grid.

**Note:** Double geometric grids have a high number of grid points in the middle or at both ends of a region. One geometrical factor for the lower (left) and upper (right) half of the region need to specified. In both cases a geometrical factor of larger than one yields a higher density of grid points at the lower end of the half and vice versa for a factor smaller than one.

Keyword arguments

- `no_of_points` int: Number of grid points in each half of the region.
- `lower_geometrical_factor` float: Geometrical factor for the lower (left) half.
- `upper_geometrical_factor` float: Geometrical factor for the upper (right) half.
Parameters

- `no_of_points` – The number of points
- `lower_geometrical_factor` – The geometrical factor for the left half
- `upper_geometrical_factor` – The geometrical factor for the right half

Returns A new `DoubleGeometricGrid` object

classmethod geometric (`no_of_points`: int = 50, `geometrical_factor`: float = 1.1)

Factory method that creates a new geometric grid.

Note: A grid that yields a varying density of grid points in the region. A geometrical factor larger than one yields a higher density of grid points at the lower end of the region and a factor smaller than one yields a higher density of grid points at the upper end of the region.

Parameters

- `no_of_points` – The number of points
- `geometrical_factor` – The geometrical factor

Returns A new `GeometricGrid` object

classmethod linear (`no_of_points`: int = 50)

Factory method that creates a new equally spaced grid.

Parameters `no_of_points` – The number of points

Returns A new `LinearGrid` object

class tc_python.diffusion.ClassicSolver

Bases: `tc_python.diffusion.Solver`

Solver using the Classic model.

Note: This solver never switches to the homogenization model even if it fails to converge. Use the `tc_python.diffusion.AutomaticSolver` if necessary instead.

`get_type` () → str

Convenience method for getting the type of the solver.

Returns The type of the solver

`set_flux_balance_equation_accuracy` (`accuracy`: float = 1e-16)

Sets the required accuracy during the solution of the flux balance equations. Default: 1.0e-16

Parameters `accuracy` – The required accuracy

Returns A new `ClassicSolver` object

`set_tieline_search_variable_to_activity` ()

Configures the solver to use the activity of a component to find the correct tie-line at the phase interface. Either activity or chemical potential are applied to reduce the degrees of freedom at the local equilibrium. Default: This is the default setting

`set_tieline_search_variable_to_potential` ()

Configures the solver to use the chemical potential of a component to find the correct tie-line at the phase interface.
interface. Either activity or chemical potential are applied to reduce the degrees of freedom at the local equilibrium. **Default:** To use the activity

**Returns** A new `ClassicSolver` object

```python
class tc_python.diffusion.ClosedSystem
    Bases: tc_python.diffusion.BoundaryCondition

    Represents a boundary for a closed system.
```

**get_type** () → `str`

Convenience method for getting the type of the boundary condition.

**Returns** The type of the boundary condition

```python
class tc_python.diffusion.CompositionProfile
    (unit_enum: tc_python.diffusion.Unit = <Unit.MASS_PERCENT: 3>)
    Bases: object

    Contains initial concentration profiles for the elements.
```

**add** (element_name: `str`, profile: `tc_python.diffusion.ElementProfile`) Adds a concentration profile for the specified element.

**Parameters**

- `element_name` – The name of the element
- `profile` – The initial concentration profile

**Returns** A `CompositionProfile` object

```python
class tc_python.diffusion.ConstantProfile
    (value: float)
    Bases: tc_python.diffusion.ElementProfile

    Represents a constant initial concentration profile.
```

**get_type** () → `str`

The type of the element profile.

**Returns** The type

```python
class tc_python.diffusion.ContinuedDiffusionCalculation
    (calculation)
    Bases: tc_python.abstract_base.AbstractCalculation

    Configuration for a diffusion calculation that is a continuation of a previous isothermal or non-isothermal diffusion calculation. It contains a subset of the settings possible in the original calculation.

    Use `set_simulation_time()` to set a simulation time that is higher than the original calculation.

    **calculate** (timeout_in_minutes: `float` = 0.0) → `tc_python.diffusion.DiffusionCalculationResult`

    Runs the diffusion calculation.

    **Parameters** `timeout_in_minutes` – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than `timeout_in_minutes`, a `UnrecoverableCalculationException` will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

    **Returns** A `DiffusionCalculationResult` which later can be used to get specific values from the calculated result
```

**set_simulation_time** (simulation_time: `float`) Sets the simulation time.

**Parameters** `simulation_time` – The simulation time [s]

**Returns** This `DiffusionIsoThermalCalculation` object
with_left_boundary_condition (boundary_condition: tc_python.diffusion.BoundaryCondition,
to: float = 1.7976931348623157e+308)

Defines the boundary condition on the left edge of the system.

**Default:** A closed-system boundary condition.

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default:** The end of the simulation.

**Note:** You can specify time-dependent boundary conditions by calling
with_left_boundary_condition() many times, with different values of the “to” parameter.

Examples:

- with_left_boundary_condition(BoundaryCondition.closed_system(), to=100)
- with_left_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C",
surface_activity), to=500)
- with_left_boundary_condition(BoundaryCondition.closed_system())

This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

**Parameters**

- **boundary_condition** – The boundary condition
- **to** – The upper time-limit for boundary_condition.

**Returns** This `DiffusionIsoThermalCalculation` object

with_options (options: tc_python.diffusion.Options, to: float = 1.7976931348623157e+308)

Sets the general simulation conditions.

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default:** The end of the simulation.

**Parameters**

- **options** – The general simulation conditions
- **to** – The upper time-limit for options.

**Returns** This `DiffusionIsoThermalCalculation` object

with_right_boundary_condition (boundary_condition: tc_python.diffusion.BoundaryCondition,
to: float = 1.7976931348623157e+308)

Defines the boundary condition on the right edge of the system.

**Default:** A closed-system boundary condition

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default:** The end of the simulation.

**Note:** You can specify time-dependent boundary conditions by calling
with_right_boundary_condition() many times, with different values of the “to” parameter.
Examples:

- `with_right_boundary_condition(BoundaryCondition.closed_system(), to=100)`
- `with_right_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C", surface_activity), to=500)`
- `with_right_boundary_condition(BoundaryCondition.closed_system())`

This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

**Parameters**

- **boundary_condition** – The boundary condition
- **to** – The upper time-limit for boundary_condition.

**Returns** This `DiffusionIsoThermalCalculation` object

**with_solver** *(solver: tc_python.diffusion.Solver, to: float = 1.7976931348623157e+308)*

Sets the solver to use (Classic, Homogenization or Automatic). **Default is Automatic.**

**Default:** The end of the simulation.

**Parameters**

- **solver** – The solver to use
- **to** – The upper time-limit for solver.

**Returns** This `DiffusionIsoThermalCalculation` object

**with_timestep_control** *(timestep_control: tc_python.diffusion.TimestepControl, to: float = 1.7976931348623157e+308)*

Sets the timestep control options.

**Default:** The end of the simulation.

**Parameters**

- **timestep_control** – The new timestep control options
- **to** – The upper time-limit for timestep_control.

**Returns** This `DiffusionIsoThermalCalculation` object

class tc_python.diffusion.DiffusionCalculationResult (result)

Bases: tc_python.abstract_base.AbstractResult

Result of a diffusion calculation. This can be used to query for specific values. For details of the axis variables, search the Thermo-Calc help.

**get_mass_fraction_at_lower_interface** *(region: str, component: str) → [typing.List[float], typing.List[float]]*

Returns the mass fraction of the specified component at the lower boundary of the specified region, in dependency of time.

**Parameters**

- **region** – The name of the region
- **component** – The name of the component
**get_mass_fraction_at_upper_interface**

```python
get_mass_fraction_at_upper_interface(region: str, component: str) → [typing.List[float], typing.List[float]]
```

Returns the mass fraction of the specified component at the upper boundary of the specified region, in dependency of time.

**Parameters**

- `region` – The name of the region
- `component` – The name of the component

**Returns**

A tuple of two lists of floats (time [s], mass fraction of the specified component)

---

**get_mass_fraction_of_component_at_time**

```python
get_mass_fraction_of_component_at_time(component: str, time: Union[tc_python.diffusion.SimulationTime, float]) → [typing.List[float], typing.List[float]]
```

Returns the mass fraction of the specified component at the specified time.

**Note:** Use the enum `tc_python.diffusion.SimulationTime` to choose the first or the last timepoint of the simulation. A timepoint close to the last one should never be specified manually because the actual end of the simulation can slightly deviate.

**Parameters**

- `component` – The name of the component
- `time` – The time [s]

**Returns**

A tuple of two lists of floats (distance [m], mass fraction of component at the specified time)

---

**get_mass_fraction_of_phase_at_time**

```python
get_mass_fraction_of_phase_at_time(phase: str, time: Union[tc_python.diffusion.SimulationTime, float]) → [typing.List[float], typing.List[float]]
```

Returns the mass fraction of the specified phase.

**Note:** Use the enum `tc_python.diffusion.SimulationTime` to choose the first or the last timepoint of the simulation. A timepoint close to the last one should never be specified manually because the actual end of the simulation can slightly deviate.

**Parameters**

- `phase` – The name of the phase
- `time` – The time [s]

**Returns**

A tuple of two lists of floats (distance [m], mass fraction of the phase at the specified time)

---

**get_mole_fraction_at_lower_interface**

```python
get_mole_fraction_at_lower_interface(region: str, component: str) → [typing.List[float], typing.List[float]]
```

Returns the mole fraction of the specified component at the lower boundary of the specified region, in dependency of time.

**Parameters**
• region – The name of the region
• component – The name of the component

Returns A tuple of two lists of floats (time [s], mole fraction of the specified component)

get_mole_fraction_at_upper_interface(region: str, component: str) → [typing.List[float], typing.List[float]]

Returns the mole fraction of the specified component at the upper boundary of the specified region, in dependency of time.

Parameters
• region – The name of the region
• component – The name of the component

Returns A tuple of two lists of floats (time [s], mole fraction of the specified component)

get_mole_fraction_of_component_at_time(component: str, time: Union[tc_python.diffusion.SimulationTime, float]) → [typing.List[float], typing.List[float]]

Returns the mole fraction of the specified component at the specified time.

Note: Use the enum tc_python.diffusion.SimulationTime to choose the first or the last timepoint of the simulation. A timepoint close to the last one should never be specified manually because the actual end of the simulation can slightly deviate.

Parameters
• component – The name of the component
• time – The time [s]

Returns A tuple of two lists of floats (distance [m], mole fraction of component at the specified time)

get_mole_fraction_of_phase_at_time(phase: str, time: Union[tc_python.diffusion.SimulationTime, float]) → [typing.List[float], typing.List[float]]

Returns the mole fraction of the specified phase.

Note: Use the enum tc_python.diffusion.SimulationTime to choose the first or the last timepoint of the simulation. A timepoint close to the last one should never be specified manually because the actual end of the simulation can slightly deviate.

Parameters
• phase – The name of the phase
• time – The time [s]

Returns A tuple of two lists of floats (distance [m], mole fraction of the phase at the specified time)

get_position_of_lower_boundary_of_region(region: str) → [typing.List[float], typing.List[float]]

Returns the position of the lower boundary of the specified region in dependency of time.
Parameters `region` – The name of the region

Returns A tuple of two lists of floats (time [s], position of lower boundary of region [m])

get_position_of_upper_boundary_of_region (`region: str`) → [typing.List[float], typing.List[float]]
Returns the position of the upper boundary of the specified region in dependency of time.

Parameters `region` – The name of the region

Returns A tuple of two lists of floats (time [s], position of upper boundary of region [m])

get_regions () → List[str]
Returns the regions of the diffusion simulation.

Note: Automatically generated regions (R_##) are included in the list.

Returns The region names

get_time_steps () → List[float]
Returns the timesteps of the diffusion simulation.

Returns The timesteps [s]

get_total_mass_fraction_of_component (`component: str`) → [typing.List[float], typing.List[float]]
Returns the total mass fraction of the specified component in dependency of time.

Parameters `component` – The name of the component

Returns A tuple of two lists of floats (time [s], total mass fraction of the component)

get_total_mass_fraction_of_component_in_phase (`component: str`, `phase: str`) → [typing.List[float], typing.List[float]]
Returns the total mass fraction of the specified component in the specified phase in dependency of time.

Parameters

- `component` – The name of the component
- `phase` – The name of the phase

Returns A tuple of two lists of floats (time [s], total mass fraction of the component in the phase)

get_total_mass_fraction_of_phase (`phase: str`) → [typing.List[float], typing.List[float]]
Returns the total mass fraction of the specified phase in dependency of the time.

Parameters `phase` – The name of the phase

Returns A tuple of two lists of floats (time [s], total mass fraction of the phase)

get_total_mole_fraction_of_component (`component: str`) → [typing.List[float], typing.List[float]]
Returns the total mole fraction of the specified component in dependency of time.

Parameters `component` – The name of the component

Returns A tuple of two lists of floats (time [s], total mole fraction of the component)

get_total_mole_fraction_of_component_in_phase (`component: str`, `phase: str`) → [typing.List[float], typing.List[float]]
Returns the total mole fraction of the specified component in the specified phase in dependency of time.

Parameters
### get_total_mole_fraction_of_phase

- **Parameters**
  - `phase` – The name of the phase

- **Returns**
  A tuple of two lists of floats (time [s], total mole fraction of the phase)

```python
get_total_mole_fraction_of_phase(phase: str) → [typing.List[float], typing.List[float]]
```

Returns the total mole fraction of the specified phase in dependency of time.

### get_total_volume_fraction_of_phase

- **Parameters**
  - `phase` – The name of the phase

- **Returns**
  A tuple of two lists of floats (time [s], total volume fraction of the phase)

```python
get_total_volume_fraction_of_phase(phase: str) → [typing.List[float], typing.List[float]]
```

Returns the total volume fraction of the specified phase in dependency of the time.

### get_values_of

- **Parameters**
  - `x_axis` – The first result quantity
  - `y_axis` – The second result quantity
  - `plot_condition` – The plot conditions
  - `independent_variable` – The independent variable

- **Returns**
  A tuple of two lists of floats (the x_axis quantity result, the y_axis quantity result) [units according to the quantities]

```python
```

Returns the specified result from the simulation, allows all possible settings.

---

**Note:** As an alternative, DICTRA Console Mode syntax can be used as well for each quantity and condition.

---

**Warning:** This is an advanced mode that is equivalent to the possibilities in the DICTRA Console Mode. Not every combination of settings will return a result.

---

### get_velocity_of_lower_boundary_of_region

- **Parameters**
  - `region` – The name of the region

- **Returns**
  A tuple of two lists of floats (time [s], velocity of lower boundary of region [m/s])

```python
get_velocity_of_lower_boundary_of_region(region: str) → [typing.List[float], typing.List[float]]
```

Returns the velocity of the lower boundary of the specified region in dependency of time.

### get_velocity_of_upper_boundary_of_region

- **Parameters**
  - `region` – The name of the region

- **Returns**
  A tuple of two lists of floats (time [s], velocity of upper boundary of region [m/s])

```python
get_velocity_of_upper_boundary_of_region(region: str) → [typing.List[float], typing.List[float]]
```

Returns the velocity of the upper boundary of the specified region in dependency of time.

---

**5.1. Calculations**
get_width_of_region (region: str) → [typing.List[float], typing.List[float]]

Returns the width of region, in dependency of time.

Parameters region – The name of the region

Returns A tuple of two lists of floats (time [s], width of the specified region [m])

save_to_disk (path: str)

Saves the result to disk. The result can later be loaded using tc_python.server.SetUp.
load_result_from_disk().

Note: The result data is represented by a whole folder containing multiple files.

Parameters path – The path to the result folder, can be relative or absolute.

Returns This DiffusionCalculationResult object

with_continued_calculation()

Returns a ContinuedDiffusionCalculation that is used for continuing a diffusion calculation
with altered settings.

Returns A ContinuedDiffusionCalculation

class tc_python.diffusion.DiffusionIsoThermalCalculation (calculation)
Bases: tc_python.abstract_base.AbstractCalculation

Configuration for an isothermal diffusion calculation.

add_console_command (console_command: str)

Registers a DICTRA Console Mode command for execution. These commands are executed after all
other configuration directly before the calculation starts to run. All commands are stored and used
until explicitly deleted using tc_python.diffusion.DiffusionIsoThermoCalculation.
remove_all_console_commands.

Parameters console_command – The DICTRA Console Mode command

Returns This DiffusionIsoThermalCalculation object

Note: It should not be necessary for most users to use this method, try to use the corresponding method
implemented in the API instead.

Warning: As this method runs raw DICTRA-commands directly in the engine, it may hang the
program in case of spelling mistakes (e.g. forgotten parenthesis, …).

add_region (region: tc_python.diffusion.Region)

Adds a region to the calculation. Regions are always added in the simulation domain from left to right.

If you want to replace an already added region, call remove_all_regions(), and add the regions
that you want to keep.

Warning: Regions must have unique names.

Parameters region – The region to be added
Returns This `DiffusionIsoThermalCalculation` object

**calculate** *(timeout_in_minutes: float = 0.0) → tc_python.diffusion.DiffusionCalculationResult*

Runs the diffusion calculation.

**Parameters** *timeout_in_minutes* – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than `timeout_in_minutes`, a `UnrecoverableCalculationException` will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

**Returns** A `DiffusionCalculationResult` which later can be used to get specific values from the calculated result

**get_system_data** () → tc_python.abstract_base.SystemData

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

---

**Note:** Parameters can only be read from unencrypted (i.e. `user`) databases loaded as `*.tdb`-file.

---

Returns The system data

**remove_all_console_commands** ()

Removes all previously added Console Mode commands.

**Returns** This `DiffusionIsoThermalCalculation` object

**remove_all_regions** ()

Removes all previously added regions.

:returns This `DiffusionIsoThermalCalculation` object

**set_simulation_time** *(simulation_time: float)*

Sets the simulation time.

**Parameters** *simulation_time* – The simulation time [s]

**Returns** This `DiffusionIsoThermalCalculation` object

**set_temperature** *(temperature: float)*

Sets the temperature for the isothermal simulation.

**Parameters** *temperature* – The temperature [K]

**Returns** This `DiffusionIsoThermalCalculation` object

**with_cylindrical_geometry** *(first_interface_position: float = 0.0)*

Sets geometry to cylindrical, corresponds to an infinitely long cylinder of a certain radius.

**Default:** A planar geometry

---

**Note:** With a cylindrical or spherical geometry, the system’s zero coordinate (left boundary) is at the centre of the cylinder or sphere by default. By specifying the `first_interface_position`, a different left-most coordinate can be defined. This allows to model a tube or a hollow sphere geometry. The highest coordinate (right boundary) is defined by the cylinder or sphere radius (i.e. by the width of all regions).

---

**Parameters** *first_interface_position* – The position of the left-most coordinate along the axis, only necessary for modeling a tube geometry [m]
with_left_boundary_condition (boundary_condition: tc_python.diffusion.BoundaryCondition, to: float = 1.7976931348623157e+308)

Defines the boundary condition on the left edge of the system.

**Default:** A closed-system boundary condition.

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default:** The end of the simulation.

**Note:** You can specify time-dependent boundary conditions by calling with_left_boundary_condition() many times, with different values of the “to” parameter.

**Examples:**

- `with_left_boundary_condition(BoundaryCondition.closed_system(), to=100)`
- `with_left_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C", surface_activity), to=500)`
- `with_left_boundary_condition(BoundaryCondition.closed_system())`

This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

**Parameters**

- **boundary_condition** – The boundary condition
- **to** – The upper time-limit for boundary_condition.

**Returns** This `DiffusionIsoThermalCalculation` object

with_options (options: tc_python.diffusion.Options, to: float = 1.7976931348623157e+308)

Sets the general simulation conditions.

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default:** The end of the simulation.

**Parameters**

- **options** – The general simulation conditions
- **to** – The upper time-limit for options.

**Returns** This `DiffusionIsoThermalCalculation` object

with_planar_geometry ()

Sets geometry to planar.

This is default.

**Returns** This `DiffusionIsoThermalCalculation` object

with_reference_state (element: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0)

The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.
All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

**Parameters**

- **element** – The name of the element
- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
- **pressure** – The pressure (in Pa) for the reference state

**Returns**

This `DiffusionIsoThermalCalculation` object

```python
with_right_boundary_condition(boundary_condition: tc_python.diffusion.BoundaryCondition, to: float = 1.7976931348623157e+308)
```

Defines the boundary condition on the right edge of the system.

**Default:** A closed-system boundary condition

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default:** The end of the simulation.

**Note:** You can specify time-dependent boundary conditions by calling `with_right_boundary_condition()` many times, with different values of the “to” parameter.

**Examples:**

- `with_right_boundary_condition(BoundaryCondition.closed_system(), to=100)`
- `with_right_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C", surface_activity), to=500)`
- `with_right_boundary_condition(BoundaryCondition.closed_system())`

This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

**Parameters**

- **boundary_condition** – The boundary condition
• **to** – The upper time-limit for boundary_condition.

**Returns** This `DiffusionIsoThermalCalculation` object

**with_solver** (`solver: tc_python.diffusion.Solver, to: float = 1.7976931348623157e+308`)
Sets the solver to use (Classic, Homogenization or Automatic). **Default is Automatic.**

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default**: The end of the simulation.

**Parameters**

• **solver** – The solver to use

• **to** – The upper time-limit for solver.

**Returns** This `DiffusionIsoThermalCalculation` object

**with_spherical_geometry** (`first_interface_position: float = 0.0`)
Sets geometry to spherical, corresponds to a sphere with a certain radius.

**Default**: A spherical geometry

**Note**: With a cylindrical or spherical geometry, the system’s zero coordinate (left boundary) is at the centre of the cylinder or sphere by default. By specifying the `first_interface_position`, a different left-most coordinate can be defined. This allows to model a tube or a hollow sphere geometry. The highest coordinate (right boundary) is defined by the cylinder or sphere radius (i.e. by the width of all regions).

**Parameters** **first_interface_position** – The position of the left-most coordinate along the axis, only necessary for modeling a hollow sphere geometry [m]

**Returns** This `DiffusionIsoThermalCalculation` object

**with_system_modifications** (`system_modifications: tc_python.abstract_base.SystemModifications`)
Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

**Note**: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `.tdb`-file.

**Parameters** **system_modifications** – The system modification to be performed

**Returns** This `DiffusionIsoThermalCalculation` object

**with_timestep_control** (`timestep_control: tc_python.diffusion.TimestepControl, to: float = 1.7976931348623157e+308`)
Sets the timestep control options.

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default**: The end of the simulation.

**Parameters**

• **timestep_control** – The new timestep control options

• **to** – The upper time-limit for timestep_control.

**Returns** This `DiffusionIsoThermalCalculation` object
**class** `tc_python.diffusion.DiffusionNonIsoThermalCalculation**(calculation)**

**Bases:** `tc_python.abstract_base.AbstractCalculation`

Configuration for a non-isothermal diffusion calculation.

**add_console_command**(console_command: str)

Registers a DICTRA Console Mode command for execution. These commands are executed after all other configuration directly before the calculation starts to run. All commands are stored and used until explicitly deleted using `tc_python.diffusion.DiffusionNonIsoThermalCalculation.remove_all_console_commands`.

**Parameters** console_command – The DICTRA Console Mode command

**Returns** This `DiffusionNonIsoThermalCalculation` object

---

**Warning:** As this method runs raw DICTRA-commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten parenthesis, ...).

**add_region**(region: `tc_python.diffusion.Region`)

Adds a region to the calculation. Regions are always added in the simulation domain from left to right.

If you want to replace an already added region, call `remove_all_regions()` and add the regions that you want to keep.

**Warning:** Regions must have unique names.

**Parameters** region – The region to be added

**Returns** This `DiffusionNonIsoThermalCalculation` object

**calculate**(timeout_in_minutes: float = 0.0) → `tc_python.diffusion.DiffusionCalculationResult`

Runs the diffusion calculation.

**Parameters** timeout_in_minutes – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than `timeout_in_minutes`, a UnrecoverableCalculationException will be thrown, the current TCPPython-block will be unusable and a new TCPPython block must be created for further calculations.

**Returns** A `DiffusionCalculationResult` which later can be used to get specific values from the calculated result

**get_system_data**() → `tc_python.abstract_base.SystemData`

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

**Note:** Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.

**Returns** The system data
remove_all_console_commands()
Removes all previously added Console Mode commands.

Returns: This DiffusionNonIsoThermalCalculation object

remove_all_regions()
Removes all previously added regions.

Returns: This DiffusionNonIsoThermalCalculation object

set_simulation_time(simulation_time: float)
Sets the simulation time.

Parameters: simulation_time – The simulation time [s]

Returns: This DiffusionNonIsoThermalCalculation object

with_cylindrical_geometry(first_interface_position: float = 0.0)
Sets geometry to cylindrical, corresponds to an infinitely long cylinder of a certain radius.

Default: A planar geometry

Note: With a cylindrical or spherical geometry, the system’s zero coordinate (left boundary) is at the centre of the cylinder or sphere by default. By specifying the first_interface_position, a different left-most coordinate can be defined. This allows to model a tube or a hollow sphere geometry. The highest coordinate (right boundary) is defined by the cylinder or sphere radius (i.e. by the width of all regions).

Parameters: first_interface_position – The position of the left-most coordinate along the axis, only necessary for modeling a tube geometry [m]

Returns: This DiffusionNonIsoThermalCalculation object

with_left_boundary_condition(boundary_condition: tc_python.diffusion.BoundaryCondition, 
to: float = 1.7976931348623157e+308)
Defines the boundary condition on the left edge of the system.

Default: A closed-system boundary condition.

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

Default: The end of the simulation.

Note: You can specify time-dependent boundary conditions by calling with_left_boundary_condition() many times, with different values of the “to” parameter.

Examples:

• with_left_boundary_condition(BoundaryCondition.closed_system(), to=100)

• with_left_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element(“C”, surface_activity), to=500)

• with_left_boundary_condition(BoundaryCondition.closed_system())

This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

Parameters
boundary_condition – The boundary condition

to – The upper time-limit for boundary_condition.

Returns This DiffusionNonIsoThermalCalculation object

with_options (options: tc_python.diffusion.Options, to: float = 1.7976931348623157e+308)
Sets the general simulation conditions.

Default: The end of the simulation.

Parameters

• options – The general simulation conditions

• to – The upper time-limit for options.

Returns This DiffusionNonIsoThermalCalculation object

with_planar_geometry ()
Sets geometry to planar.

This is default.

Returns This DiffusionNonIsoThermalCalculation object

with_reference_state (element: str, phase: str = ‘SER’, temperature: float = -1.0, pressure: float = 100000.0)
The reference state for a component is important when calculating activities, chemical potentials and
enthalpies and is determined by the database being used. For each component the data must be referred to
a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different
databases can use different reference states for the same element/component. It is important to be careful
when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by
the database. If the reference state in the database is not suitable for your purposes, use this command to
set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set
as default for a major component in alloys dominated by the component). In such cases, the temperature
and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition
as an end member of the phase. The reference state is an end member of a phase. The selection of the end
member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the
end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

Parameters

• element – The name of the element

• phase – Name of a phase used as the new reference state. Or SER for the Stable Element
  Reference.

• temperature – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE
  which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.

• pressure – The pressure (in Pa) for the reference state

Returns This DiffusionNonIsoThermalCalculation object
with_right_boundary_condition(boundary_condition: tc_python.diffusion.BoundaryCondition, to: float = 1.7976931348623157e+308)

Defines the boundary condition on the right edge of the system.

**Default:** A closed-system boundary condition

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default:** The end of the simulation.

**Note:** You can specify time-dependent boundary conditions by calling `with_right_boundary_condition()` many times, with different values of the “to” parameter.

Examples:

- `with_right_boundary_condition(BoundaryCondition.closed_system(), to=100)`
- `with_right_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C", surface_activity), to=500)`
- `with_right_boundary_condition(BoundaryCondition.closed_system())`

This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

**Parameters**

- `boundary_condition` – The boundary condition
- `to` – The upper time-limit for boundary_condition.

**Returns** This `DiffusionNonIsoThermalCalculation` object

with_solver(solver: tc_python.diffusion.Solver, to: float = 1.7976931348623157e+308)

Sets the solver to use (Classic, Homogenization or Automatic). **Default is Automatic.**

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default:** The end of the simulation.

**Parameters**

- `solver` – The solver to use
- `to` – The upper time-limit for solver.

**Returns** This `DiffusionNonIsoThermalCalculation` object

with_spherical_geometry(first_interface_position: float = 0.0)

Sets geometry to spherical, corresponds to a sphere with a certain radius.

**Default:** A spherical geometry

**Note:** With a cylindrical or spherical geometry, the system’s zero coordinate (left boundary) is at the centre of the cylinder or sphere by default. By specifying the `first_interface_position`, a different left-most coordinate can be defined. This allows to model a tube or a hollow sphere geometry. The highest coordinate (right boundary) is defined by the cylinder or sphere radius (i.e. by the width of all regions).

**Parameters** `first_interface_position` – The position of the left-most coordinate along the axis, only necessary for modeling a hollow sphere geometry [m]
Returns This *DiffusionNonIsothermalCalculation* object

**with_system_modifications** *(system_modifications: tc_python.abstract_base.SystemModifications)*

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

**Note:** This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a *.tdb*-file.

**Parameters**

- **system_modifications** – The system modification to be performed

**Returns** This *DiffusionNonIsothermalCalculation* object

**with_temperature_profile** *(temperature_profile: tc_python.utils.TemperatureProfile)*

Sets the temperature profile to use with this calculation.

**Parameters**

- **temperature_profile** – The temperature profile object (specifying time / temperature points)

**Returns** This *DiffusionNonIsothermalCalculation* object

**with_timestep_control** *(timestep_control: tc_python.diffusion.TimestepControl, to: float = 1.7976931348623157e+308)*

Sets the timestep control options.

It is possible specify the upper time-point for which this setting is valid using the parameter “to”.

**Default:** The end of the simulation.

**Parameters**

- **timestep_control** – The new timestep control options
- **to** – The upper time-limit for timestep_control.

**Returns** This *DiffusionNonIsothermalCalculation* object

**class** *tc_python.diffusion.DoubleGeometricGrid*(no_of_points: int = 50, lower_geometrical_factor: float = 1.1, upper_geometrical_factor: float = 0.9)

**Bases:** *tc_python.diffusion.CalculatedGrid*

Represents a double geometric grid.

**get_lower_geometrical_factor** () → float

Returns the lower geometrical factor (for the left half).

**Returns** The lower geometrical factor

**get_no_of_points** () → int

Returns number of grid points.

**Returns** The number of grid points

**get_type** () → str

Type of the grid.

**Returns** The type of the grid

**get_upper_geometrical_factor** ()

Returns the upper geometrical factor (for the right half).
Returns The upper geometrical factor

```
set_lower_geometrical_factor(geometrical_factor: float = 1.1)
```

Sets the lower (left half) geometrical factor.

**Note:** A geometrical factor of larger than one yields a higher density of grid points at the lower end of the half and vice versa for a factor smaller than one.

**Parameters**
- `geometrical_factor` – The geometrical factor for the left half

**Returns** This `DoubleGeometricGrid` object

```
set_no_of_points(no_of_points: int = 50)
```

Sets the number of grid points.

**Parameters**
- `no_of_points` – The number of points

**Returns** This `DoubleGeometricGrid` object

```
set_upper_geometrical_factor(geometrical_factor: float = 0.9)
```

Sets the upper (right half) geometrical factor.

**Note:** A geometrical factor of larger than one yields a higher density of grid points at the lower end of the half and vice versa for a factor smaller than one.

**Parameters**
- `geometrical_factor` – The geometrical factor for the right half

**Returns** This `DoubleGeometricGrid` object

```python
class tc_python.diffusion.ElementProfile
    Bases: tc_python.diffusion.AbstractElementProfile

    Factory class providing objects for configuring a step, function or linear initial concentration profile.

    **classmethod constant** (value: float)
        Factory method that creates a new constant initial concentration profile.

        **Parameters**
        - `value` – The constant composition in the region. [unit as defined in `CompositionProfile`].

        **Returns** A new `ConstantProfile` object

    **classmethod funct** (dictra_console_mode_function: str)
        Factory method that creates a new initial concentration profile defined by a function in DICTRA Console Mode syntax.

        **Parameters**
        - `dictra_console_mode_function` – The function, expressed in DICTRA Console Mode syntax.

        **Returns** A new `FunctionProfile` object

    **Note:** This is an advanced feature, preferably a complex concentration profile should be generated using third party libraries and added to the simulation using `tc_python.diffusion.PointByPointGrid`.
```
classmethod linear(start_value: float, end_value: float)

Factory method that creates a new linear initial concentration profile.

Parameters

• start_value – Composition at the left side of the region [unit as defined in CompositionProfile].

• end_value – Composition at the right side of the region [unit as defined in CompositionProfile].

Returns A new LinearProfile object

classmethod step(lower_boundary: float, upper_boundary: float, step_at: float)

Factory method that creates a new initial concentration profile with a step at the specified distance, otherwise the composition is constant at the specified values.

Parameters

• lower_boundary – Composition before the step [unit as defined in CompositionProfile].

• upper_boundary – Composition after the step [unit as defined in CompositionProfile].

• step_at – The distance where the step should be [m].

Returns A new StepProfile object

class tc_python.diffusion.FixFluxValue

Bases: tc_python.diffusion.BoundaryCondition

get_type() → str

The type of the boundary condition.

Returns The type

set_flux(element_name: str, J: str = '0', to_time: float = 1.7976931348623157e+308)

Enter functions that yield the flux times the molar volume for the specified element. May be a function of time, temperature and pressure: J(T,P,TIME).

Parameters

• element_name – The name of the element

• J – the function J(T,P,TIME)

• to_time – The max-time for which the flux function is used.

class tc_python.diffusion.FixedCompositions

Bases: tc_python.diffusion.BoundaryCondition

Represents a boundary having fixed composition values.

get_type() → str

The type of the boundary condition.

Returns The type

set_composition(element_name: str, value: float)

Sets the composition for the specified element.

Note: The boundary composition needs to be specified for each element.
Parameters

- **element_name** – The name of the element
- **value** – The composition value [unit according to the constructor parameter]

```python
class tc_python.diffusion.FunctionProfile (dictra_console_mode_function: str)
    Bases: tc_python.diffusion.ElementProfile

    Creates an initial concentration profile defined by a function in DICTRA Console Mode syntax.
```

**Note:** This is an advanced feature, preferably a complex concentration profile should be generated using third party libraries and added to the simulation using `tc_python.diffusion.PointByPointGrid`.

```python
get_type () → str

    The type of the element profile.
```

**Returns** The type

```python
class tc_python.diffusion.GeneralLowerHashinShtrikman
    Bases: tc_python.diffusion.HomogenizationFunctions

    General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

    Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
```

```python
class tc_python.diffusion.GeneralLowerHashinShtrikmanExcludedPhase (excluded_phases: List[str] = [])
    Bases: tc_python.diffusion.HomogenizationFunctions

    General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

    Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

    The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
```

```python
class tc_python.diffusion.GeneralUpperHashinShtrikman
    Bases: tc_python.diffusion.HomogenizationFunctions

    General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

    Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
```

```python
class tc_python.diffusion.GeneralUpperHashinShtrikmanExcludedPhase (excluded_phases: List[str] = [])
    Bases: tc_python.diffusion.HomogenizationFunctions

    General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

    Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

    The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
```
```python
    Bases: tc_python.diffusion.CalculatedGrid

    Represents a geometric grid.

    get_geometrical_factor() → float
        Returns the geometrical factor.
        Returns The geometrical factor

    get_no_of_points() → int
        Returns the number of grid points.
        Returns The number of grid points

    get_type() → str
        Returns the type of grid.
        Returns The type

    set_geometrical_factor(geometrical_factor: float = 1.1)
        Sets the geometrical factor.
        Note: A geometrical factor larger than one yields a higher density of grid points at the lower end of the region and a factor smaller than one yields a higher density of grid points at the upper end of the region.
        Parameters geometrical_factor – The geometrical factor
        Returns This GeometricGrid object

    set_no_of_points(no_of_points: int = 50)
        Sets the number of grid points.
        Parameters no_of_points – The number of points
        Returns This GeometricGrid object

class tc_python.diffusion.GridPoint(distance: float):
    Bases: object

    Represents a grid point, this is used in combination with grids of the type tc_python.diffusion.PointByPointGrid.

    add_composition(element: str, value: float)
        Adds a composition for the specified element to the grid point.
        Parameters
            • element – The element
            • value – The composition value [unit as defined for the grid]
        Returns This GridPoint object

class tc_python.diffusion.HashinShtrikmanBoundMajority
    Bases: tc_python.diffusion.HomogenizationFunctions

    Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.
    Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
```

5.1. Calculations
class tc_python.diffusion.HashinShtrikmanBoundMajorityExcludedPhase(excluded_phases: List[str] = [])

Bases: tc_python.diffusion.HomogenizationFunctions

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

class tc_python.diffusion.HashinShtrikmanBoundPrescribed(matrix_phase: str)

Bases: tc_python.diffusion.HomogenizationFunctions

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.


Bases: tc_python.diffusion.HomogenizationFunctions

class tc_python.diffusion.HomogenizationFunction(value)

Bases: enum.Enum

Homogenization function used for the homogenization solver. Many homogenization functions are based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. Default: RULE_OF_MIXTURES (i.e. upper Wiener bounds)

GENERAL_LOWER_HASHIN_SHTRIKMAN = 0

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

GENERAL_UPPER_HASHIN_SHTRIKMAN = 1

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

HASHIN_SHTRIKMAN_BOUND_MAJORITY = 2

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

INVERSE_RULE_OF_MIXTURES = 4

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion

RULE_OF_MIXTURES = 3

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion

class tc_python.diffusion.HomogenizationFunctions

Bases: object

classmethod general_lower_hashin_shtrikman()

Factory method that creates a new homogenization function of the type
General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

**Returns** A new `GeneralLowerHashinShtrikman` object

**classmethod general_lower_hashin_shtrikman_excluded_phase**

Factory method that creates a new homogenization function of the type `GeneralLowerHashinShtrikmanExcludedPhase`.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

**Parameters** `excluded_phases` – The excluded phases

**Returns** A new `GeneralLowerHashinShtrikmanExcludedPhase` object

**classmethod general_upper_hashin_shtrikman_excluded_phase**

Factory method that creates a new homogenization function of the type `GeneralUpperHashinShtrikmanExcludedPhase`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

**Parameters** `excluded_phases` – The excluded phases

**Returns** A new `GeneralUpperHashinShtrikmanExcludedPhase` object

**classmethod hashin_shtrikman_bound_majority**

Factory method that creates a new homogenization function of the type `HashinShtrikmanBoundMajority`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

**Returns** A new `HashinShtrikmanBoundMajority` object
classmethod hashin_shtrikman_bound_majority_excluded_phase(excluded_phases: List[str] = [])

Factory method that creates a new homogenization function of the type HashinShtrikmanBoundMajorityExcludedPhase.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

Parameters excluded_phases – The excluded phases

Returns A new HashinShtrikmanBoundMajorityExcludedPhase object

classmethod hashin_shtrikman_bound_prescribed(matrix_phase: str)

Factory method that creates a new homogenization function of the type HashinShtrikmanBoundPrescribed.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

Parameters matrix_phase – The matrix phase

Returns A new HashinShtrikmanBoundPrescribed object

classmethod hashin_shtrikman_bound_prescribed_excluded_phase(matrix_phase: str, excluded_phases: List[str] = [])

Factory method that creates a new homogenization function of the type HashinShtrikmanBoundPrescribedExcludedPhase.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

Parameters

• matrix_phase – The matrix phase
• excluded_phases – The excluded phases

Returns A new HashinShtrikmanBoundPrescribedExcludedPhase object

classmethod inverse_rule_of_mixtures()

Factory method that creates a new homogenization function of the type InverseRuleOfMixtures.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

Returns A new InverseRuleOfMixtures object

classmethod inverse_rule_of_mixtures_excluded_phase(excluded_phases: List[str] = [])

Factory method that creates a new homogenization function of the type InverseRuleOfMixturesExcludedPhase.
Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

**Parameters** `excluded_phases` – The excluded phases

**Returns** A new `InverseRuleOfMixturesExcludedPhase` object

classmethod `labyrinth_factor_f` *(matrix_phase: str)*

Factory method that creates a new homogenization function of the type `LabyrinthFactorF`.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

**Parameters** `matrix_phase` – The matrix phase

**Returns** A new `LabyrinthFactorF` object

classmethod `labyrinth_factor_f2` *(matrix_phase: str)*

Factory method that creates a new homogenization function of the type `LabyrinthFactorF2`.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

**Parameters** `matrix_phase` – The matrix phase

**Returns** A new `LabyrinthFactorF2` object

classmethod `rule_of_mixtures`()

Factory method that creates a new homogenization function of the type `RuleOfMixtures`.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

**Returns** A new `RuleOfMixtures` object

classmethod `rule_of_mixtures_excluded_phase` *(excluded_phases: List[str] = [])*

Factory method that creates a new homogenization function of the type `RuleOfMixturesExcludedPhase`.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

**Parameters** `excluded_phases` – The excluded phases

**Returns** A new `RuleOfMixturesExcludedPhase` object

class `tc_python.diffusion.HomogenizationSolver`

**Bases**: `tc_python.diffusion.Solver`

Solver using the *Homogenization model*.

---

**Note**: This solver always uses the homogenization model, even if all regions have only one phase. The solver is **significantly slower than the Classic model**. Use the `tc_python.diffusion.AutomaticSolver` instead if you do not need that behavior.

**disable_global_minimization()**

Disables global minimization to be used in equilibrium calculations. **Default**: Disabled
**Note:** In general, using global minimization **significantly increases the simulation time**, but there is also a significantly reduced risk for non-converged equilibrium calculations.

**Returns** A new `HomogenizationSolver` object

### disable_interpolation_scheme()
Configures the simulation not use any interpolation scheme. **Default:** To use the logarithmic interpolation scheme with 10000 discretization steps

**Note:** The homogenization scheme can be switched on by using `with_linear_interpolation_scheme` or `with_logarithmic_interpolation_scheme`.

### enable_global_minimization()
Enables global minimization to be used in equilibrium calculations. **Default:** Disabled

**Note:** In general, using global minimization **significantly increases the simulation time**, but there is also a significantly reduced risk for non-converged equilibrium calculations.

**Returns** A new `HomogenizationSolver` object

### get_type() → str
The type of solver.

**Returns** The type

### set_fraction_of_free_memory_to_use(fraction: float)
Sets the maximum fraction of free physical memory to be used by the interpolation scheme. **Default:** 1 / 10 of the free physical memory

**Parameters**

- `fraction` – The maximum free physical memory fraction to be used

**Returns** A new `HomogenizationSolver` object

### set_memory_to_use(memory_in_megabytes: float)
Sets the maximum physical memory in megabytes to be used by the interpolation scheme. **Default:** 1000 MBytes of the free physical memory

**Parameters**

- `memory_in_megabytes` – The maximum physical memory to be used

**Returns** A new `HomogenizationSolver` object

### with_function(homogenization_function: tc_python.diffusion.HomogenizationFunctions)
Sets the homogenization function used by the homogenization model.

**Parameters**

- `homogenization_function` – The homogenization function used by the homogenization model

**Returns** A new `HomogenizationSolver` object

### with_linear_interpolation_scheme(steps: int = 10000)
Configures the simulation to use the linear interpolation scheme. **Default:** To use the logarithmic interpolation scheme with 10000 discretization steps

**Parameters**

- `steps` – The number of discretization steps in each dimension

**Returns** A new `HomogenizationSolver` object
with_logarithmic_interpolation_scheme (steps: int = 10000)
Configures the simulation to use the linear interpolation scheme. Default: To use the logarithmic interpolation scheme with 10000 discretization steps

Parameters steps – The number of discretization steps in each dimension

Returns A new HomogenizationSolver object

class tc_python.diffusion.InverseRuleOfMixtures
Bases: tc_python.diffusion.HomogenizationFunctions

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

class tc_python.diffusion.InverseRuleOfMixturesExcludedPhase (excluded_phases: List[str] = [])
Bases: tc_python.diffusion.HomogenizationFunctions

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

Excluded phases are not considered in the diffusion calculations.

class tc_python.diffusion.LabyrinthFactorF (matrix_phase: str)
Bases: tc_python.diffusion.HomogenizationFunctions

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

class tc_python.diffusion.LabyrinthFactorF2 (matrix_phase: str)
Bases: tc_python.diffusion.HomogenizationFunctions

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

class tc_python.diffusion.LinearGrid (no_of_points: int = 50)
Bases: tc_python.diffusion.CalculatedGrid

Represents an equally spaced grid.

get_no_of_points () \rightarrow int
Returns the number of grid points.

get_type () \rightarrow str
Type of the grid.

Returns The type

set_no_of_points (no_of_points: int = 50)
Sets the number of grid points.

Parameters no_of_points – The number of points

Returns This LinearGrid object

class tc_python.diffusion.LinearProfile (start_value: float, end_value: float)
Bases: tc_python.diffusion.ElementProfile

Represents a linear initial concentration profile.

get_type () \rightarrow str
The type of the element profile.
Returns The type

class tc_python.diffusion.MixedZeroFluxAndActivity
    Bases: tc_python.diffusion.BoundaryCondition

Represents a boundary having zero-flux as well as fixed-activity conditions.

Default: On that boundary for every element without an explicitly defined condition, a zero-flux boundary condition is used.

get_type() → str
    The type of the boundary condition.

Returns The type

set_activity_for_element(element_name: str, activity: str, to_time: float = 1.7976931348623157e+308)
    Sets an activity expression for an element at the boundary. Enter a formula that the software evaluates during the calculation.
    
The formula can be:
    • a function of the variable TIME
    • a constant

The formula must be written with these rules:
    • a number must begin with a number (not a .)
    • a number must have a dot or an exponent (E)

The operators +,-,*,/,** (exponentiation) can be used and with any level of parenthesis. As shown, the following operators must be followed by open and closed parentheses ()
    • SQRT(X) is the square root
    • EXP(X) is the exponential
    • LOG(X) is the natural logarithm
    • LOG10(X) is the base 10 logarithm
    • SIN(X), COS(X), TAN(X), ASIN(X), ACOS(X), ATAN(X)
    • SINH(X), COSH(X), TANH(X), ASINH(X), ACOSH(X), ATANH(X)
    • SIGN(X)
    • ERF(X) is the error function

Default: the expression entered is used for the entire simulation.

Parameters
    • element_name – The name of the element
    • activity – The activity
    • to_time – The max-time for which the activity is used.

set_zero_flux_for_element(element_name: str)
    Sets a zero-flux condition for an element at the boundary. Default for all elements at the boundary without an explicitly defined condition

Parameters element_name – The name of the element
class tc_python.diffusion.Options
    Bases: object

    General simulation conditions for the diffusion calculations.

disable_forced_starting_values_in_equilibrium_calculations()
    Disables forced starting values for the equilibrium calculations. The default is ‘enable_automatic_forced_starting_values_in_equilibrium_calculations’.

    Returns This Options object
disable_save_results_to_file()
    Disables the saving of results to file during the simulation. Default: Saving of the results at every timestep

    Returns This Options object
enable_automatic_forced_starting_values_in_eq_calculations()
    Lets calculation engine decide if forced start values for the equilibrium calculations should be used. This is the default setting.

    Returns This Options object
enable_forced_starting_values_in_equilibrium_calculations()
    Enables forced start values for the equilibrium calculations. The default is ‘enable_automatic_forced_starting_values_in_equilibrium_calculations’.

    Returns This Options object
enable_save_results_to_file(every_nth_step: int = -1)
    Enables and configures saving of results to file during the simulation. They can be saved for every n-th or optionally for every timestep (-1). Default: Saving of the results at every timestep

    Parameters every_nth_step -- -1 or a value ranging from 0 to 99

    Returns This Options object
enable_time_integration_method_automatic()
    Enables automatic selection of integration method. This is the default method.

    Returns This Options object
enable_time_integration_method_euler_backwards()
    Enables Euler backwards integration. The default method is enable_time_integration_method_automatic.

    Note: This method is more stable but less accurate and may be necessary if large fluctuations occur in the profiles.

    Returns This Options object
enable_time_integration_method_trapezoidal()
    Enables trapezoidal integration.

    Note: If large fluctuations occur in the profiles, it may be necessary to use the more stable but less accurate Euler backwards method.

    Returns This Options object
set_default_driving_force_for_phases_allowed_to_form_at_interf(driving_force: float = 1e-05)

Sets the default required driving force for phases allowed to form at the interfaces. Default: 1.0e-5

Note: The required driving force (evaluated as $DGM(ph)$) is used for determining whether an inactive phase is stable, i.e. actually formed. $DGM$ represents the driving force normalized by $RT$ and is dimensionless.

Parameters driving_force – The driving force ($DGM(ph)$) [-]

Returns This Options object

class tc_python.diffusion.PointByPointGrid(unit_enum: tc_python.diffusion.Unit = <Unit.MASS_PERCENT: 3>)

Bases: tc_python.diffusion.AbstractGrid

Represents a point-by-point grid. This is setting the grid and the compositions at once, it is typically used to enter a measured composition profile or the result from a previous calculation.

Note: If a point-by-point grid is used, it is not necessary to specify the grid and composition profile separately.

add_point(grid_point: tc_python.diffusion.GridPoint)

Adds a grid point to the grid.

Parameters grid_point – The grid point

Returns This PointByPointGrid object

get_type() → str

Type of the grid.

Returns The type

class tc_python.diffusion.Region(name: str)

Bases: object

Represents a region of the simulation domain that can contain more that one phase.

Note: The first added phase represents the matrix phase, while all later added phases are spheriod phases, i.e. precipitate phases.

add_phase(phase_name: str, is_matrix_phase: bool = False)

Adds a phase to the region, each region must contain at least one phase.

Note: Normally the matrix phase and the precipitate phases are automatically chosen based on the presence of all profile elements in the phase and if it has diffusion data. If multiple phases have equal properties, the phase that was added first is chosen. The matrix phase can be explicitly set by using is_matrix_phase=True.

Note: If multiple phases are added to a region, the homogenization model is applied. That means that average properties of the local phase mixture are used.
Parameters

- **phase_name** – The phase name
- **is_matrix_phase** – If set to *True* this phase is explicitly set as matrix phase for the region, if no phase is set to *True*, the matrix phase is chosen automatically

Returns This `Region` object

**add_phase_allowed_to_form_at_left_interface**(*phase_name*: str, *driving_force*: float = 1e-05)

Adds a phase allowed to form at the left boundary of the region (an *inactive phase*). The phase will only appear at the interface as a new automatic region if the driving force to form it is sufficiently high.

Parameters

- **phase_name** – The phase name
- **driving_force** – The driving force for the phase to form (*DGM*(ph))

Returns This `Region` object

**add_phase_allowed_to_form_at_right_interface**(*phase_name*: str, *driving_force*: float = 1e-05)

Adds a phase allowed to form at the right boundary of the region (an *inactive phase*). The phase will only appear at the interface as a new automatic region if the driving force to form it is sufficiently high.

Parameters

- **phase_name** – The phase name
- **driving_force** – The driving force for the phase to form (*DGM*(ph))

Returns This `Region` object

**remove_all_phases**()

Removes all previously added phases from the region.

Returns This `Region` object

**set_width**(*width*: float)

Defined the width of the region.

**Note:** This method needs only to be used if a calculated grid has been defined (using `with_grid()`).

**Parameters**

- **width** – The width [m]

Returns This `Region` object

**with_composition_profile**(*initial_compositions*: tc_python.diffusion.CompositionProfile)

Defines the initial composition profiles for all elements in the region.

**Note:** This method needs only to be used if a calculated grid has been defined (using `with_grid()`).

**Parameters**

- **initial_compositions** – The initial composition profiles for all elements

Returns This `Region` object
with_grid(grid: tc_python.diffusion.CalculatedGrid)
Defines a calculated grid in the region. If measured composition profiles or the result from a previous calculation should be used, instead with_point_by_point_grid_containing_compositions() needs to be applied.

**Note:** The composition profiles need to be defined separately using with_composition_profile(), additionally the region width needs to be specified using set_width().

**Parameters**
- **grid** – The grid

**Returns**
This Region object

with_point_by_point_grid_containing_compositions(grid: tc_python.diffusion.PointByPointGrid)
Defines a point-by-point grid in the region. This is setting the grid and the compositions at once, it is typically used to enter a measured composition profile or the result from a previous calculation. If the composition profile should be calculated (linear, geometric, ...) with_grid() should be used instead.

**Note:** If a point-by-point grid is used, with_grid(), with_composition_profile() and set_width() are unnecessary and must not be used.

**Parameters**
- **grid** – The point-by-point grid

**Returns**
This Region object

class tc_python.diffusion.RuleOfMixtures
Bases: tc_python.diffusion.HomogenizationFunctions

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

class tc_python.diffusion.RuleOfMixturesExcludedPhase(excluded_phases: List[str] = [])
Bases: tc_python.diffusion.HomogenizationFunctions

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

Excluded phases are not considered in the diffusion calculations.

class tc_python.diffusion.SimulationTime(value)
Bases: enum.Enum

Specifying special time steps for the evaluation of diffusion results.

**Note:** These placeholders should be used because especially the actual last timestep will slightly differ from the specified end time of the simulation.

**FIRST = 0**
Represents the first timestep of the simulation

**LAST = 1**
Represents the last timestep of the simulation
class tc_python.diffusion.Solver
    Bases: tc_python.diffusion.AbstractSolver

Factory class providing objects representing a solver.

classmethod automatic()
    Factory method that creates a new automatic solver. This is the default solver and recommended for most applications.

    Note: This solver uses the homogenization model if any region has more than one phase, otherwise it uses the classic model.

    Returns A new AutomaticSolver object

classmethod classic()
    Factory method that creates a new classic solver.

    Note: This solver never switches to the homogenization model even if the solver fails to converge. Use the tc_python.diffusion.AutomaticSolver if necessary instead.

    Returns A new ClassicSolver object

classmethod homogenization()
    Factory method that creates a new homogenization solver.

    Note: This solver always uses the homogenization model, even if all regions have only one phase. The solver is significantly slower than the Classic model. Use the tc_python.diffusion.AutomaticSolver instead if you do not need that behavior.

    Returns A new HomogenizationSolver object

class tc_python.diffusion.StepProfile(lower_boundary: float, upper_boundary: float, step_at: float)
    Bases: tc_python.diffusion.ElementProfile

Represents an initial constant concentration profile with a step at the specified position.

    get_type() \rightarrow \text{str}
        The type of the element profile.

        Returns The type

class tc_python.diffusion.TimestepControl
    Bases: object

Settings that control the time steps in the simulation.

    disable_check_interface_position()
        Disables checking of the interface position, i.e. the timesteps are not controlled by the phase interface displacement during the simulation. The default setting is disable_automatic_check_interface_position.

        Returns This TimestepControl object

5.1. Calculations
enable_automatic_check_interface_position()

   Lets calculation engine decide if checking of the interface position should be used. This is the default setting.

   Returns This TimestepControl object

enable_check_interface_position()

   Enables checking of the interface position, i.e. the timesteps are controlled by the phase interface displacement during the simulation. The default setting is :func:`enable_automatic_check_interface_position`.

   Returns This TimestepControl object

set_initial_time_step (initial_time_step: float = 1e-07)

   Sets the initial timestep. Default: 1.0e-7 s

   Parameters initial_time_step – The initial timestep [s]

   Returns This TimestepControl object

set_max_absolute_error (absolute_error: float = 1e-05)

   Sets the maximum absolute error. Default: 1.0e-5

   Parameters absolute_error – The maximum absolute error

   Returns This TimestepControl object

set_max_relative_error (relative_error: float = 0.05)

   Sets the maximum relative error. Default: 0.05

   Parameters relative_error – The maximum relative error

   Returns This TimestepControl object

set_max_timestep_allowed_as_percent_of_simulation_time (max_timestep_allowed_as_percent_of_simulation_time: float = 10.0)

   The maximum timestep allowed during the simulation, specified in percent of the simulation time. Default: 10.0%

   Parameters max_timestep_allowed_as_percent_of_simulation_time – The maximum timestep allowed [%]

   Returns This TimestepControl object

set_max_timestep_increase_factor (max_timestep_increase_factor: float = 2.0)

   Sets the maximum timestep increase factor. Default: 2

   Parameters max_timestep_increase_factor – The maximum timestep increase factor

   Returns This TimestepControl object

set_smallest_time_step_allowed (smallest_time_step_allowed: float = 1e-07)

   Sets the smallest time step allowed during the simulation. This is required when using the automatic procedure to determine the time step. Default: 1.0e-7 s

   Parameters smallest_time_step_allowed – The smallest timestep allowed [s]

   Returns This TimestepControl object

class tc_python.diffusion.Unit (value)

   Bases: enum.Enum

Note: For example, if 2 is entered the maximum time step is twice as long as the previous time step taken.
Represents a composition unit.

\[
\begin{align*}
\text{MASS\_FRACTION} &= 2 \\
&\quad \text{Mass fraction.} \\
\text{MASS\_PERCENT} &= 3 \\
&\quad \text{Mass percent.} \\
\text{MOLE\_FRACTION} &= 0 \\
&\quad \text{Mole fraction.} \\
\text{MOLE\_PERCENT} &= 1 \\
&\quad \text{Mole percent.} \\
\text{U\_FRACTION} &= 4 \\
&\quad \text{U fraction}
\end{align*}
\]

5.1.7 Module “propertymodel”

```python
class tc_python.propertymodel.PropertyModelCalculation(calculator):
    Bases: tc_python.abstract_base.AbstractCalculation

    Configuration for a Property Model calculation.

    Note: Specify the settings, the calculation is performed with `calculate()`.
```

```python
def add_poly_command(poly_command: str)
    Registers a POLY Console Mode command for execution. These commands are executed after all other configuration directly before the calculation starts to run. All commands are stored and used until explicitly deleted using `remove_all_poly_commands`.

    Parameters poly_command – The POLY Console Mode command

    Returns This `PropertyModelCalculation` object
```

Warning: As this method runs raw POLY-commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten parenthesis, ...).

```python
def calculate(timeout_in_minutes: float = 0.0) → tc_python.propertymodel.PropertyModelResult
    Runs the Property Model calculation.

    Parameters timeout_in_minutes – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a CalculationEngineException will be thrown.

    Returns A `PropertyModelResult` which later can be used to get specific values from the simulation.
```

```python
def get_argument_default(argument_id: str) → object
    Returns the default value for the specified argument. The argument id can be obtained with `get_arguments()`.
```
**Parameters** argument_id – The argument id

**Returns** The default value (the type depends on the argument)

```python
def get_argument_description(argument_id: str) -> str:
    Returns the detailed description of the argument. The id can be obtained with `get_arguments()`.
```

**Parameters** argument_id – The argument id

**Returns** The detailed description

```python
def get_arguments() -> Set[str]:
    Returns a list of the arguments of the Property Model.
```

**Note:** The arguments are the ‘UI-panel components’ defined in the Property Model interface method `provide_ui_panel_components()`. They have the same id as specified in the Property Model. The naming is different because there is no UI present.

**Returns** The ids of the available arguments

```python
def get_dynamic_arguments() -> Set[str]:
    Returns a list of the dynamic arguments of the Property Model.
```

**Note:** Dynamic arguments are “extra” arguments created by pressing the “plus” button that can occur next to the UI-panel for some models, when running the Property Model from within Thermo-Calc. You can use them also from the API by `invoke_dynamic_argument()`.

**Returns** The ids of the available dynamic arguments

```python
def get_model_description() -> str:
    Returns the description text of the current model.
```

**Returns** The description

```python
def get_model_parameter_value(model_parameter_id: str) -> float:
    Returns the current value of an optimizable model parameter. The id can be obtained with `get_model_parameters()`.
```

**Parameters** model_parameter_id – The model parameter id

**Returns** The current value [unit according to the parameter meaning]

```python
def get_model_parameters() -> Set[str]:
    Returns a list of the optimizable model parameters.
```

**Note:** The model parameters are an optional set of variables that can be used within the Property Model. Typically they are used to provide the possibility to inject parameter values during an optimization into the model. This allows the dynamic development of Property Models that need to be fitted to experimental data. The model parameters are controlled with the Property Model interface methods `provide_model_parameters` and `set_model_parameter`.

**Returns** The ids of the optimizable model parameters
get_system_data () → tc_python.abstract_base.SystemData
Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using with_system_modifications().

Note: Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb*-file.

Returns The system data

invoke_dynamic_argument (argument_id: str)
Increases the number of instances of this dynamic argument by one, the argument will have an id such as argument_1, argument_2, ... if the dynamic argument is called argument.

Note: You can obtain all available dynamic arguments by using get_dynamic_arguments().

Parameters argument_id – argument_id: The argument id

Returns This PropertyModelCalculation object

remove_all_conditions ()
Removes all set classic POLY conditions.

Note: This does not affect the compositions set by set_composition().

Returns This PropertyModelCalculation object

remove_all_poly_commands ()
Removes all previously added POLY Console Mode commands.

Returns This PropertyModelCalculation object

remove_dependent_element ()
Removes a manually set dependent element. This method does not affect the automatic choice of the dependent element if set_composition() is used.

Returns This PropertyModelCalculation object

set_argument (argument: str, value: str)
Sets the specified model argument to the specified value. The id can be obtained with get_arguments().

Parameters
  • argument – The argument id
  • value – The value [unit according to the argument meaning]

Returns This PropertyModelCalculation object

set_composition (element_name: str, value: float)
Sets the composition of a element. The unit for the composition can be changed using set_composition_unit().

Default: Mole percent (CompositionUnit.MOLE_PERCENT)

Parameters
**element_name** – The element

**value** – The composition value [composition unit defined for the calculation]

**Returns** This `PropertyModelCalculation` object

```python
def set_composition_unit(unit_enum: tc_python.utils.CompositionUnit = <CompositionUnit.MOLE_PERCENT: 2>)
```

Sets the composition unit.

**Default:** Mole percent (`CompositionUnit.MOLE_PERCENT`).

**Parameters**

- **unit_enum** – The new composition unit

**Returns** This `PropertyModelCalculation` object

```python
def set_condition(classic_condition: str, value: float)
```

Adds a classic POLY condition. If that method is used, all conditions need to be specified in such a way. If this method is used, it is necessary to set the dependent element manually using `set_dependent_element()`.

**Default if not specified:** pressure $P = 1e5$ Pa, system size $N = 1$, Temperature $T = 1000$ K

**Warning:** It is not possible to mix POLY-commands and compositions using `set_composition()`.

**Note:** It should not be necessary for most users to use this method, try to use `set_composition()` instead.

**Warning:** As this method runs raw POLY-commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten parenthesis,…).

**Parameters**

- **classic_condition** – The classic POLY condition (for example: $X(CR)$)

- **value** – The value of the condition

**Returns** This `PropertyModelCalculation` object

```python
def set_dependent_element(dependent_element_name: str)
```

Sets the dependent element manually.

**Note:** It should not be necessary for most users to use this method. Setting the dependent element manually is only necessary and allowed if `set_condition()` is used.

**Parameters**

- **dependent_element_name** – The name of the dependent element

**Returns** This `PropertyModelCalculation` object

```python
def set_model_parameter(model_parameter_id: str, value)
```

Resets an optimizable model parameter. The id can be obtained with `get_model_parameters()`.

**Parameters**
• **model_parameter_id** – The model parameter id
• **value** – The new value of the parameter

Returns This `PropertyModelCalculation` object

```python
def set_temperature(temperature: float = 1000)
    Sets the temperature.
    Default: 1000 K
```

Parameters

- **temperature** – The temperature [K]

Returns This `PropertyModelCalculation` object

```python
def with_system_modifications(system_modifications: tc_python.abstract_base.SystemModifications)
    Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a *.tdb* file.
```

Parameters

- **system_modifications** – The system modification to be performed

Returns This `PropertyModelCalculation` object

```python
class tc_python.propertymodel.PropertyModelResult(result)
    Bases: tc_python.abstract_base.AbstractResult

    The result of a Property Model calculation.

    ```python
    def get_result_quantities() -> Set[str]
        Returns a list of the available result quantities defined in the Property Model.

        Returns The ids of the defined result quantities
    ```

    ```python
    def get_result_quantity_description(result_quantity_id: str) -> str
        Returns the detailed description of the result quantity. The id can be obtained by get_result_quantities().

        Parameters **result_quantity_id** – The result quantity id

        Returns The detailed description
    ```

    ```python
    def get_single_equilibrium_result(result_quantity_id: str = '') -> tc_python.single_equilibrium.SingleEquilibriumResult
        Returns a result quantity value. The available result quantities can be obtained by get_result_quantities().

        Parameters **result_quantity_id** – The id of the result quantity.

        Returns The requested value [unit depending on the quantity], if the result is a SingleEquilibriumResult, is returned.
    ```

    ```python
    def get_value_of(result_quantity_id: str) -> Union[float, Dict[str, float]]
        Returns a result quantity value. The available result quantities can be obtained by get_result_quantities().

        Parameters **result_quantity_id** – The id of the result quantity

        Returns The requested value [unit depending on the quantity]. If the result is parameterized, parameter-value pairs are returned.
```
save_to_disk(path: str)
Saves the result to disk. The result can later be loaded using `tc_python.server.SetUp.load_result_from_disk()`.

**Note:** The result data is represented by a whole folder possibly containing multiple files.

**Parameters**
- **path** – The path to the result folder, can be relative or absolute.
- **Returns** This `PropertyModelResult` object

### 5.1.8 Module “material_to_material”

```python
class tc_python.material_to_material.AbstractConstantCondition
    Bases: object
    The abstract base class for all constant conditions.

class tc_python.material_to_material.AbstractMaterialToMaterialCalculationAxis
    Bases: object
    The abstract base class of all calculation axis.

class tc_python.material_to_material.ConstantCondition
    Bases: tc_python.material_to_material.AbstractConstantCondition
    A constant condition.

classmethod fraction_of_material_b(fraction_of_material_b: float = 0.5)
    Creates a constant fraction of material B condition object.

    **Note:** The unit depends on the composition unit setting in the calculator object.

    **Parameters**
        - **fraction_of_material_b** – The fraction of material B [weight-fraction or mole-fraction]
    **Returns** The condition object

classmethod temperature(temperature: float = 1000)
    Creates a constant temperature condition object.

    **Parameters**
        - **temperature** – The temperature [K]
    **Returns** The condition object

class tc_python.material_to_material.FractionOfMaterialBAxis(from_fraction:
    float = 0.0, to_fraction: float = 1.0, start_fraction:
    float = 0.5)
    Bases: tc_python.material_to_material.MaterialToMaterialCalculationAxis
    A fraction of material B axis.

class tc_python.material_to_material.FractionOfMaterialBCondition(fraction_of_material_b:
    float = 0.5)
    Bases: tc_python.material_to_material.ConstantCondition
```
A constant fraction of material B condition.

```python
class tc_python.material_to_material.MaterialToMaterialCalculationAxis:
    Bases: tc_python.material_to_material.AbstractMaterialToMaterialCalculationAxis
    A calculation axis.

    @classmethod
    def fraction_of_material_b(cls, from_fraction: float = 0.0, to_fraction: float = 1.0, start_fraction: float = 0.5):
        Creates a fraction of material B axis object.

        Parameters
        * from_fraction – The left axis limit [weight-fraction or mole-fraction]
        * to_fraction – The right axis limit [weight-fraction or mole-fraction]
        * start_fraction – The start fraction of the calculation [weight-fraction or mole-fraction]

        Returns
        A new `FractionOfMaterialBAxis` axis object
```

```python
class tc_python.material_to_material.MaterialToMaterialCalculationContainer:
    Bases: object
    Provides access to the calculation objects for all Material to Material calculations.
    These are specialised calculations for mixtures of two materials A and B. Otherwise they behave identical to the
    corresponding regular single equilibrium, property diagram and phase diagram calculations.

    with_phase_diagram_calculation:
        (default_conditions: bool = True, components: List[str] = [],)

    Creates a Material to Material phase diagram (map) calculation.

    Parameters
    * default_conditions – If True, automatically sets the conditions \( N=1 \) and \( P=100000 \)
    * components – Specify here the components of the system (for example: \([\text{AL2O3}, \ldots]\),
                  only necessary if they differ from the elements. If this option is used, all elements of
                  the system need to be replaced by a component.

    Returns
    A new `MaterialToMaterialPhaseDiagramCalculation` object
```
with_property_diagram_calculation

```python
default_conditions: bool = True, components: List[str] = []
```

Creates a Material to Material property diagram (step) calculation.

**Parameters**

- **default_conditions** — If True, automatically sets the conditions $N=1$ and $P=100000$

- **components** — Specify here the components of the system (for example: `[AL2O3, ...]`), *only necessary if they differ from the elements*. If this option is used, all elements of the system need to be replaced by a component.

**Returns** A new `MaterialToMaterialPropertyDiagramCalculation` object

with_single_equilibrium_calculation

```python
default_conditions: bool = True, components: List[str] = []
```

Creates a Material to Material single equilibrium calculation.

**Parameters**

- **default_conditions** — If True, automatically sets the conditions $N=1$ and $P=100000$

- **components** — Specify here the components of the system (for example: `[AL2O3, ...]`), *only necessary if they differ from the elements*. If this option is used, all elements of the system need to be replaced by a component.

**Returns** A new `MaterialToMaterialSingleEquilibriumCalculation` object

class tc_python.material_to_material.MaterialToMaterialPhaseDiagramCalculation

```python
calculator
```

Configuration for a Material to Material phase diagram calculation.

**Note:** Specify the conditions, the calculation is performed with `calculate()`.

add_initial_equilibrium

```python
(initial_equilibrium: tc_python.step_or_map_diagrams.InitialEquilibrium)
```

Add initial equilibrium start points from which a phase diagram is calculated.

Scans along the axis variables and generates start points when the scan procedure crosses a phase boundary. It may take a little longer to execute than using the minimum number of start points, as some lines may be calculated more than once. But the core remembers all node points and subsequently stops calculations along a line when it finds a known node point.

It is also possible to create a sequence of start points from one initial equilibria.

**Parameters** **initial_equilibrium** — The initial equilibrium

**Returns** This `MaterialToMaterialPhaseDiagramCalculation` object

calculate

```python
(keep_previous_results: bool = False, timeout_in_minutes: float = 0.0)
```

Performs the phase diagram calculation.

**Warning:** If you use `keep_previous_results=True`, you must not use another calculator or even get results in between the calculations using `calculate()`. Then the previous results will actually be lost.
Parameters

- **keep_previous_results** – If True, results from any previous call to this method are appended. This can be used to combine calculations with multiple start points if the mapping fails at a certain condition.

- **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

**Returns** A new `MaterialToMaterialPhaseDiagramResult` object which later can be used to get specific values from the calculated result.

disable_global_minimization()
Disables global minimization.

**Default:** Enabled

**Returns** This `MaterialToMaterialPhaseDiagramCalculation` object

dont_keep_default_equilibria()
Do not keep the initial equilibria added by default.

This is only relevant in combination with `add_initial_equilibrium()`.

This is the default behavior.

**Returns** This `MaterialToMaterialPhaseDiagramCalculation` object

enable_global_minimization()
Enables global minimization.

**Default:** Enabled

**Returns** This `MaterialToMaterialPhaseDiagramCalculation` object

get_components() → List[str]
Returns the names of the components in the system (including all components auto-selected by the database(s)).

**Returns** The component names

get_gibbs_energy_addition_for(phase: str) → float
Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters** phase – Specify the name of the (stoichiometric or solution) phase with the addition

**Returns** Gibbs energy addition to G per mole formula unit.

get_system_data() → tc_python.abstract_base.SystemData
Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

**Note:** Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.

**Returns** The system data
keep_default_equilibria()
Keep the initial equilibria added by default. This is only relevant in combination with
add_initial_equilibrium().

Default behavior is to not keep default equilibria.

Returns This MaterialToMaterialPhaseDiagramCalculation object

remove_all_initial_equilibria()
Removes all previously added initial equilibria.

Returns This MaterialToMaterialPhaseDiagramCalculation object

run_poly_command(command: str)
Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Parameters command – The Thermo-Calc Console Mode command

Returns This MaterialToMaterialPhaseDiagramCalculation object

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

set_activities(activities: Dict[str, float])
Sets the constant activity conditions.

Note: The activity conditions are identical for both materials.

Parameters activities – The constant activities

Returns This MaterialToMaterialPhaseDiagramCalculation object

set_composition_unit(unit: tc_python.utils.CompositionUnit = <Compositio-
Unit.MASS_PERCENT: 0>)
Sets the composition unit of both materials A and B.

Default: Weight percent

Parameters unit – The composition unit of both materials A and B

Returns This MaterialToMaterialPhaseDiagramCalculation object

set_gibbs_energy_addition_for(phrase: str, gibbs_energy: float)
Used to specify the additional energy term (always being a constant) of a given phase. The value (gibbs_energy) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

Parameters

• phase – Specify the name of the (stoichiometric or solution) phase with the addition

• gibbs_energy – Addition to G per mole formula unit
Returns This `MaterialToMaterialPhaseDiagramCalculation` object

### set_material_a (**composition**: Dict[str, float], **dependent_component**: str = None)
Sets the composition of the material A.
The unit is set with `set_composition_unit()`.

Tip: The material can also have constant activity conditions, they are set in `set_activities()`.

Parameters
- **composition** – The composition of the material A
- **dependent_component** – The dependent component of the material A

Returns This `MaterialToMaterialPhaseDiagramCalculation` object

### set_material_b (**composition**: Dict[str, float], **dependent_component**: str = None)
Sets the composition of the material B.
The unit is set with `set_composition_unit()`.

Tip: The material can also have constant activity conditions, they are set in `set_activities()`.

Parameters
- **composition** – The composition of the material B
- **dependent_component** – The dependent component of the material B

Returns This `MaterialToMaterialPhaseDiagramCalculation` object

### set_phase_to_dormant (**phase**: str)
Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

Parameters **phase** – The phase name or `ALL_PHASES` for all phases

Returns This `MaterialToMaterialPhaseDiagramCalculation` object

### set_phase_to_entered (**phase**: str, **amount**: float = 1.0)
Sets the phase to the status ENTERED, that is the default state.

Parameters
- **phase** – The phase name or `ALL_PHASES` for all phases
- **amount** – The phase fraction (between 0.0 and 1.0)

Returns This `MaterialToMaterialPhaseDiagramCalculation` object

### set_phase_to_fixed (**phase**: str, **amount**: float)
Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

Parameters
- **phase** – The phase name
- **amount** – The fixed phase fraction (between 0.0 and 1.0)
Returns This MaterialToMaterialPhaseDiagramCalculation object

set_phase_to_suspended (phase: str)
Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

Parameters phase – The phase name or ALL_PHASES for all phases

Returns This MaterialToMaterialPhaseDiagramCalculation object

set_pressure (pressure: float)
Sets the pressure (i.e. the condition \( P \)).

Note: If the flag default_conditions=True has been set during the creation of the calculator, the pressure is set to 1000 hPa by default.

Parameters pressure – The pressure [Pa]

Returns This MaterialToMaterialPhaseDiagramCalculation object

set_system_size (system_size: float)
Sets the system size (i.e. the condition ‘N’, the number of moles).

Note: If the flag default_conditions=True has been set during the creation of the calculator, the system size is set to 1.0 moles by default.

Parameters system_size – The system size [mole]

Returns This MaterialToMaterialPhaseDiagramCalculation object

with_first_axis (axis: tc_python.material_to_material.MaterialToMaterialCalculationAxis)
Sets the first axis (either temperature of fraction of material B). This calculation type requires that both temperature and fraction of material B axis are set.

Parameters axis – The axis

Returns This MaterialToMaterialPhaseDiagramCalculation object

with_options (options: tc_python.step_or_map_diagrams.PhaseDiagramOptions)
Sets the simulation options.

Parameters options – The simulation options

Returns This PhaseDiagramCalculation object

with_reference_state (component: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0)
The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set
as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

Parameters

- **component** – The name of the element must be given.
- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
- **pressure** – The Pressure (in Pa) for the reference state.

Returns This MaterialToMaterialPhaseDiagramCalculation object

with_second_axis (axis: tc_python.material_to_material.MaterialToMaterialCalculationAxis)

Sets the second axis (either temperature of fraction of material B). This calculation type requires that both temperature and fraction of material B axis are set.

Parameters **axis** – The axis

Returns This MaterialToMaterialPhaseDiagramCalculation object

with_system_modifications (system_modifications: tc_python.abstract_base.SystemModifications)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a *.tdb*-file.

Parameters **system_modifications** – The system modification to be performed

Returns This MaterialToMaterialPhaseDiagramCalculation object

class tc_python.material_to_material.MaterialToMaterialPhaseDiagramResult (result)

Bases: tc_python.step_or_map_diagrams.PhaseDiagramResult

Result of a Material to Material phase diagram calculation, it can be evaluated using quantities or Console Mode syntax.

add_coordinate_for_phase_label (x: float, y: float)

Sets a coordinate in the result plot for which the stable phases will be evaluated and provided in the result data object. This can be used to plot the phases of a region into the phase diagram or just to programmatically evaluate the phases in certain regions.

Warning: This method takes coordinates of the plot axes and not of the calculation axis.

Parameters
• **x** – The coordinate of the first plot axis (“x-axis”) [unit of the plot axis]
• **y** – The coordinate of the second plot axis (“y-axis”) [unit of the plot axis]

**Returns** This `MaterialToMaterialPhaseDiagramResult` object

```python
def get_values_grouped_by_quantity_of(x_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], y_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str]) -> tc_python.step_or_map_diagrams.PhaseDiagramResultValues
```

Returns x-y-line data grouped by the multiple datasets of the specified quantities (for example in dependency of components). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`. Usually the result data represents the phase diagram.

**Note:** The different datasets will contain `NaN`-values between different subsections and are not sorted (because they are unsortable due to their nature).

**Note:** It’s possible to use functions as axis variables, either by using `ThermodynamicQuantity.user_defined_function`, or by using an expression that contains ‘=’.

**Example**
```python
def get_values_grouped_by_quantity_of(’T’, ThermodynamicQuantity.user_defined_function(’HM.T’))
```

**Example**
```python
def get_values_grouped_by_quantity_of(’T’, ’CP=HM.T’)
```

**Parameters**

• **x_quantity** – The first quantity (“x-axis”), Console Mode syntax strings can be used as an alternative (for example ‘T’), `MATERIAL_B_FRACTION`, or even a function (for example ‘f=T*1.01’)

• **y_quantity** – The second quantity (“y-axis”), Console Mode syntax strings can be used as an alternative (for example ‘NV’), `MATERIAL_B_FRACTION`, or even a function (for example ‘CP=HM.T’)

**Returns** The phase diagram data

```python
def get_values_grouped_by_stable_phases_of(x_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], y_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str]) -> tc_python.step_or_map_diagrams.PhaseDiagramResultValues
```

Returns x-y-line data grouped by the sets of “stable phases” (for example “LIQUID” or “LIQUID + FCC_A1”). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`. Usually the result data represents the phase diagram.

**Note:** The different datasets will contain `NaN`-values between different subsections and are not sorted (because they are unsortable due to their nature).

**Note:** It’s possible to use functions as axis variables, either by using `ThermodynamicQuantity.user_defined_function`, or by using an expression that contains ‘=’.

**Example**
```python
def get_values_grouped_by_stable_phases_of(’T’, ThermodynamicQuantity.user_defined_function(’HM.T’))
```
Example get_values_grouped_by_quantity_of("T", "CP=HM.T")

Parameters

- **x_quantity** – The first quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example ‘T’), MATERIAL_B_FRACTION, or even a function (for example ‘f=T*1.01’)

- **y_quantity** – The second quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example ‘NV’), MATERIAL_B_FRACTION, or even a function (for example ‘CP=HM.T’)

Returns The phase diagram data

```
remove_phase_labels()  
```

Erasers all added coordinates for phase labels.

**Returns** This MaterialToMaterialPhaseDiagramResult object

```
save_to_disk(path: str)  
```

Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with load_result_from_disk()

**Parameters** path – the path to the folder you want the result to be saved in. It can be relative or absolute.

**Returns** this MaterialToMaterialPhaseDiagramResult object

```
set_phase_name_style(phase_name_style_enum: tc_python.step_or_map_diagrams.PhaseNameStyle = <PhaseNameStyle.NONE: 0>)  
```

Sets the style of the phase name labels that will be used in the result data object (constitution description, ordering description, ...).

**Default:** PhaseNameStyle.NONE

**Parameters** phase_name_style_enum – The phase name style

**Returns** This MaterialToMaterialPhaseDiagramResult object

```
class tc_python.material_to_material.MaterialToMaterialPropertyDiagramCalculation(calculator)  
```

**Bases:** tc_python.step_or_map_diagrams.AbstractPropertyDiagramCalculation

Configuration for a Material to Material property diagram calculation.

**Note:** Specify the conditions and possibly other settings, the calculation is performed with calculate().

```
calculate(keep_previous_results: bool = False, timeout_in_minutes: float = 0.0) →  
tc_python.material_to_material.MaterialToMaterialPropertyDiagramResult  
```

Performs the Material to Material property diagram calculation.

**Warning:** If you use `keep_previous_results=True`, you must not use another calculator or even get results in between the calculations using `calculate()`. Then the previous results will actually be lost.

**Parameters**
- **keep_previous_results** – If True, results from any previous call to this method are appended. This can be used to combine calculations with multiple start points if the stepping fails at a certain condition.

- **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

**Returns** A new `MaterialToMaterialPropertyDiagramResult` object which later can be used to get specific values from the calculated result.

```python
disable_global_minimization()
```
Disables global minimization.

**Default:** Enabled

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

```python
disable_step_separate_phases()
```
Disables *step separate phases*. This is the default setting.

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

```python
enable_global_minimization()
```
Enables global minimization.

**Default:** Enabled

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

```python
enable_step_separate_phases()
```
Enables *step separate phases*.

**Default:** By default separate phase stepping is disabled

**Note:** This is an advanced option, it is used mostly to calculate how the Gibbs energy for a number of phases varies for different compositions. This is particularly useful to calculate Gibbs energies for complex phases with miscibility gaps and for an ordered phase that is never disordered (e.g. SIGMA-phase, G-phase, MU-phase, etc.).

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

```python
get_components() \(\rightarrow\) List[str]
```
Returns the names of the components in the system (including all components auto-selected by the database(s)).

**Returns** The component names

```python
get_gibbs_energy_addition_for(\text{phase}: \text{str}) \rightarrow \text{float}
```
Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters** phase – Specify the name of the (stoichiometric or solution) phase with the addition

**Returns** Gibbs energy addition to G per mole formula unit.
**get_system_data** () → *tc_python.abstract_base.SystemData*

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

**Note:** Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb*-file.

**Returns** The system data

**run_poly_command** (*command: str*)

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

**Parameters** *command* – The Thermo-Calc Console Mode command

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

**Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

**Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

**set_activities** (*activities: Dict[str, float]*)

Sets the constant activity conditions.

**Note:** The activity conditions are identical for both materials.

**Parameters** *activities* – The constant activities

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

**set_composition_unit** (*unit: tc_python.utils.CompositionUnit = <CompositionUnit.MASS_PERCENT: 0>*)

Sets the composition unit of both materials A and B.

**Default:** Weight percent

**Parameters** *unit* – The composition unit of both materials A and B

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

**set_gibbs_energy_addition_for** (*phase: str, gibbs_energy: float*)

Used to specify the additional energy term (always being a constant) of a given phase. The value (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

**Parameters**

- *phase* – Specify the name of the (stoichiometric or solution) phase with the addition
- *gibbs_energy* – Addition to G per mole formula unit

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object
set_material_a (composition: Dict[str, float], dependent_component: str = None)

Sets the composition of the material A.

The unit is set with set_composition_unit().

**Tip:** The material can also have constant activity conditions, they are set in set_activities().

**Parameters**

- **composition** – The composition of the material A
- **dependent_component** – The dependent component of the material A

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

set_material_b (composition: Dict[str, float], dependent_component: str = None)

Sets the composition of the material B.

The unit is set with set_composition_unit().

**Tip:** The material can also have constant activity conditions, they are set in set_activities().

**Parameters**

- **composition** – The composition of the material B
- **dependent_component** – The dependent component of the material B

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

set_phase_to_dormant (phase: str)

Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

**Parameters** **phase** – The phase name or ALL_PHASES for all phases

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

set_phase_to_entered (phase: str, amount: float = 1.0)

Sets the phase to the status ENTERED, that is the default state.

**Parameters**

- **phase** – The phase name or ALL_PHASES for all phases
- **amount** – The phase fraction (between 0.0 and 1.0)

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object

set_phase_to_fixed (phase: str, amount: float)

Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

**Parameters**

- **phase** – The phase name
- **amount** – The fixed phase fraction (between 0.0 and 1.0)

**Returns** This `MaterialToMaterialPropertyDiagramCalculation` object
**set_phase_to_suspended** *(phase: str)*

Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

**Parameters** *phase* – The phase name or ALL_PHASES for all phases

**Returns** This MaterialToMaterialPropertyDiagramCalculation object

**set_pressure** *(pressure: float)*

Sets the pressure (i.e. the condition \( P \)).

**Note:** If the flag default_conditions=True has been set during the creation of the calculator, the pressure is set to 1000 hPa by default.

**Parameters** *pressure* – The pressure [Pa]

**Returns** This MaterialToMaterialPropertyDiagramCalculation object

**set_system_size** *(system_size: float)*

Sets the system size (i.e. the condition ‘N’, the number of moles).

**Note:** If the flag default_conditions=True has been set during the creation of the calculator, the system size is set to 1.0 moles by default.

**Parameters** *system_size* – The system size [mole]

**Returns** This MaterialToMaterialPropertyDiagramCalculation object

**with_axis** *(axis: tc_python.material_to_material.MaterialToMaterialCalculationAxis)*

Sets the axis (either temperature of fraction of material B). This calculation type requires that either temperature or fraction of material B is set as a constant condition - the other one is set as an axis.

**Parameters** *axis* – The axis

**Returns** This MaterialToMaterialPropertyDiagramCalculation object

**with_constant_condition** *(condition: tc_python.material_to_material.ConstantCondition)*

Sets the constant condition (either temperature of fraction of material B). This calculation type requires that either temperature or fraction of material B is set as a constant condition - the other one is set as an axis.

**Parameters** *condition* – The condition

**Returns** This MaterialToMaterialPropertyDiagramCalculation object

**with_options** *(options: tc_python.step_or_map_diagrams.PropertyDiagramOptions)*

Sets the simulation options.

**Parameters** *options* – The simulation options

**Returns** This MaterialToMaterialPropertyDiagramCalculation object

**with_reference_state** *(component: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0)*

The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.
All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

**Parameters**

- **component** – The name of the element must be given.
- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
- **pressure** – The Pressure (in Pa) for the reference state.

**Returns**

This `MaterialToMaterialPropertyDiagramCalculation` object

with `system_modifications` (system_modifications: tc_python.abstract_base.SystemModifications)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

**Note:** This is only possible if the system has been read from unencrypted (i.e. `user`) databases loaded as a *.tdb*-file.

**Parameters**

- **system_modifications** – The system modification to be performed

**Returns**

This `MaterialToMaterialPropertyDiagramCalculation` object

class tc_python.material_to_material.MaterialToMaterialPropertyDiagramResult (result)

Bases: tc_python.step_or_map_diagrams.PropertyDiagramResult

Result of a Material to Material property diagram. It can be used to query for specific values.

**get_values_grouped_by_quantity_of** (x_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], y_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], sort_and_merge: bool = True) → Dict[str, tc_python.utils.ResultSetValueGroup]

Returns x-y-line data grouped by the multiple datasets of the specified quantities (typically the phases). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`.

**Note:** The different datasets might contain `NaN`-values between different subsections and might not be sorted even if the flag `sort_and_merge` has been set (because they might be unsortable due to their nature).
Note: It's possible to use functions as axis variables, either by using `ThermodynamicQuantity`, `user_defined_function`, or by using an expression that contains `='.

Example

```
get_values_grouped_by_quantity_of('T', ThermodynamicQuantity.user_defined_function('HM.T'))
```

Example

```
get_values_grouped_by_quantity_of('T', 'CP=HM.T')
```

Parameters

- **x_quantity** – The first quantity (“x-axis”), Console Mode syntax strings can be used as an alternative (for example ‘T’), `MATERIAL_B_FRACTION`, or even a function (for example ‘f=T*1.01’)

- **y_quantity** – The second quantity (“y-axis”), Console Mode syntax strings can be used as an alternative (for example ‘NV’), `MATERIAL_B_FRACTION`, or even a function (for example ‘CP=HM.T’)

- **sort_and_merge** – If `True`, the data is sorted and merged into as few subsections as possible (divided by `NaN`)

Returns Containing the datasets with the quantities as their keys

```
get_values_grouped_by_stable_phases_of(x_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], y_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], sort_and_merge: bool = True) → Dict[str, tc_python.utils.ResultValueGroup]
```

Returns x-y-line data grouped by the sets of “stable phases” (for example “LIQUID” or “LIQUID + FCC_A1”). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`.

Note: The different datasets might contain `NaN`-values between different subsections and different lines of an ambiguous dataset. They might not be sorted even if the flag `sort_and_merge` has been set (because they might be unsortable due to their nature).

Parameters

- **x_quantity** – The first quantity (“x-axis”), Console Mode syntax strings can be used as an alternative (for example ‘T’), `MATERIAL_B_FRACTION`, or even a function (for example ‘f=T*1.01’)

- **y_quantity** – The second quantity (“y-axis”), Console Mode syntax strings can be used as an alternative (for example ‘NV’), `MATERIAL_B_FRACTION`, or even a function (for example ‘CP=HM.T’)

5.1. Calculations
sort_and_merge – If True, the data will be sorted and merged into as few subsections as possible (divided by NaN)

Returns: Containing the datasets with the quantities as their keys

get_values_of(x_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], y_quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str]) → [typing.List[float], typing.List[float]]

Returns sorted x-y-line data without any separation. Use get_values_grouped_by_quantity_of() or get_values_grouped_by_stable_phases_of() instead if you need such a separation. The available quantities can be found in the documentation of the factory class ThermodynamicQuantity.

Note: This method will always return sorted data without any NaN-values. If it is unsortable that might give data that is hard to interpret. In such a case you need to choose the quantity in another way or use one of the other methods. One example of this is to use quantities with All-markers, for example MassFractionOfAComponent("All").

Note: It's possible to use functions as axis variables, either by using ThermodynamicQuantity.user_defined_function(), or by using an expression that contains ‘=’.

Example

get_values_grouped_by_quantity_of('T', ThermodynamicQuantity.user_defined_function('HM.T'))

Example get_values_grouped_by_quantity_of('T', 'CP=HM.T')

Parameters

• x_quantity – The first thermodynamic quantity (“x-axis”), Console Mode syntax strings can be used as an alternative (for example ‘T’, MATERIAL_B_FRACTION, or even a function (for example ‘f=T*1.01’).

• y_quantity – The second thermodynamic quantity (“y-axis”), Console Mode syntax strings can be used as an alternative (for example ‘NV’), MATERIAL_B_FRACTION, or even a function (for example ‘CP=HM.T’)

Returns: A tuple containing the x- and y-data in lists

save_to_disk(path: str)

Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with load_result_from_disk()

Parameters path – the path to the folder you want the result to be saved in. It can be relative or absolute.

Returns this MaterialToMaterialPropertyDiagramResult object

set_phase_name_style(phase_name_style_enum: tc_python.step_or_map_diagrams.PhaseNameStyle = <PhaseNameStyle.NONE: 0>)

Sets the style of the phase name labels that will be used in the result data object (constitution description, ordering description, ...).

Default: PhaseNameStyle.NONE

Parameters phase_name_style_enum – The phase name style

Returns: This MaterialToMaterialPropertyDiagramResult object
class tc_python.material_to_material.MaterialToMaterialSingleEquilibriumCalculation(calculator):

Bases: tc_python.single_equilibrium.AbstractSingleEquilibriumCalculation

Configuration for a Material to Material single fraction of B calculation.

Note: Specify the conditions and possibly other settings, the calculation is performed with `calculate()`.

calculate(timeout_in_minutes: float = 0.0) → tc_python.material_to_material.MaterialToMaterialSingleEquilibriumResult

Performs the material to material calculation.

Note: The calculation result is no temporary result object.

Parameters timeout_in_minutes – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

Returns A new MaterialToMaterialSingleEquilibriumResult object which can be used to get specific values from the calculated result. It is undefined behavior to use that object after the state of the calculation has been changed.

disable_global_minimization()  

Turns the global minimization completely off.

Returns This MaterialToMaterialSingleEquilibriumCalculation object

enable_global_minimization()  

Turns the global minimization on (using the default settings).

Returns This MaterialToMaterialSingleEquilibriumCalculation object

get_components() → List[str]

Returns a list of components in the system (including all components auto-selected by the database(s)).

Returns The components

get_gibbs_energy_addition_for(phase: str) → float

Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

Parameters phase – Specify the name of the (stoichiometric or solution) phase with the addition

Returns Gibbs energy addition to G per mole formula unit.

get_system_data() → tc_python.abstract_base.SystemData

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

Note: Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.

Returns The system data
**run_poly_command** (*command: str*)

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

**Parameters**
- **command** – The Thermo-Calc Console Mode command

**Returns**
This `MaterialToMaterialSingleEquilibriumCalculation` object

**Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

**Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

**set_activities** (*activities: Dict[str, float]*)

Sets the constant activity conditions.

**Note:** The activity conditions are identical for both materials.

**Parameters**
- **activities** – The constant activities

**Returns**
This `MaterialToMaterialSingleEquilibriumCalculation` object

**set_component_to_entered** (*component: str*)

Sets the specified component to the status ENTERED, that is the default state.

**Parameters**
- **component** – The component name or `ALL_COMPONENTS`

**Returns**
This `MaterialToMaterialSingleEquilibriumCalculation` object

**set_component_to_suspended** (*component: str, reset_conditions: bool = False*)

Sets the specified component to the status SUSPENDED, i.e. it is ignored in the calculation.

**Parameters**
- **reset_conditions** – if ‘True’ also remove composition conditions for the component if they are defined
- **component** – The component name or `ALL_COMPONENTS`

**Returns**
This `MaterialToMaterialSingleEquilibriumCalculation` object

**set_composition_unit** (*unit: tc_python.utils.CompositionUnit = <CompositionUnit.MASS_PERCENT: 0>*)

Sets the composition unit of both materials A and B.

**Default:** Weight percent

**Parameters**
- **unit** – The composition unit of both materials A and B

**Returns**
This `MaterialToMaterialSingleEquilibriumCalculation` object

**set_gibbs_energy_addition_for** (*phase: str, gibbs_energy: float*)

Used to specify the additional energy term (always being a constant) of a given phase. The value (`gibbs_energy`) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.
Parameters

• **phase** – Specify the name of the (stoichiometric or solution) phase with the addition

• **gibbs_energy** – Addition to G per mole formula unit

Returns This *MaterialToMaterialSingleEquilibriumCalculation* object

**set_material_a** *(composition: Dict[str, float], dependent_component: str = None)*

Sets the composition of the material A.

The unit is set with *set_composition_unit()*.

Tip: The material can also have constant activity conditions, they are set in *set_activities()*.

Parameters

• **composition** – The composition of the material A

• **dependent_component** – The dependent component of the material A

Returns This *MaterialToMaterialSingleEquilibriumCalculation* object

**set_material_b** *(composition: Dict[str, float], dependent_component: str = None)*

Sets the composition of the material B.

The unit is set with *set_composition_unit()*.

Tip: The material can also have constant activity conditions, they are set in *set_activities()*.

Parameters

• **composition** – The composition of the material B

• **dependent_component** – The dependent component of the material B

Returns This *MaterialToMaterialSingleEquilibriumCalculation* object

**set_phase_to_dormant** *(phase: str)*

Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

Parameters **phase** – The phase name or *ALL_PHASES* for all phases

Returns This *MaterialToMaterialSingleEquilibriumCalculation* object

**set_phase_to_entered** *(phase: str, amount: float = 1.0)*

Sets the phase to the status ENTERED, that is the default state.

Parameters

• **phase** – The phase name or *ALL_PHASES* for all phases

• **amount** – The phase fraction (between 0.0 and 1.0)

Returns This *MaterialToMaterialSingleEquilibriumCalculation* object

**set_phase_to_fixed** *(phase: str, amount: float)*

Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.
Parameters

- **phase** – The phase name
- **amount** – The fixed phase fraction (between 0.0 and 1.0)

Returns This `MaterialToMaterialSingleEquilibriumCalculation` object

```python
set_phase_to_suspended(phase: str)
```
Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

Parameters **phase** – The phase name or `ALL PHASES` for all phases

Returns This `MaterialToMaterialSingleEquilibriumCalculation` object

```python
set_pressure(pressure: float)
```
Sets the pressure (i.e. the condition $P$).

Note: If the flag `default_conditions=True` has been set during the creation of the calculator, the pressure is set to 1000 hPa by default.

Parameters **pressure** – The pressure [Pa]

Returns This `MaterialToMaterialSingleEquilibriumCalculation` object

```python
set_system_size(system_size: float)
```
Sets the system size (i.e. the condition ‘N’, the number of moles).

Note: If the flag `default_conditions=True` has been set during the creation of the calculator, the system size is set to 1.0 moles by default.

Parameters **system_size** – The system size [mole]

Returns This `MaterialToMaterialSingleEquilibriumCalculation` object

```python
with_first_constant_condition(condition: tc_python.material_to_material.ConstantCondition)
```
Sets the first constant condition (either temperature of fraction of material B).

Parameters **condition** – The condition

Returns This `MaterialToMaterialSingleEquilibriumCalculation` object

```python
with_options(options: tc_python.single_equilibrium.SingleEquilibriumOptions)
```
Sets the simulation options.

Parameters **options** – The simulation options

Returns This `SingleEquilibriumCalculation` object

```python
with_reference_state(component: str, phase: str = 'SER', temperature: float = -1.0, pressure: float = 100000.0)
```
The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.
By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

**Parameters**

- **component** – The name of the element must be given.
- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
- **pressure** – The Pressure (in Pa) for the reference state.

**Returns** This `MaterialToMaterialSingleEquilibriumCalculation` object

### with_second_constant_condition

Sets the second constant condition (either temperature of fraction of material B).

**Parameters** condition – The condition

**Returns** This `MaterialToMaterialSingleEquilibriumCalculation` object

### with_system_modifications

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

**Parameters** system_modifications – The system modification to be performed

**Returns** This `MaterialToMaterialSingleEquilibriumCalculation` object

---

**Note:** This is only possible if the system has been read from unencrypted (i.e. `user`) databases loaded as a *.tdb*-file.

---

**Parameters** system_modifications – The system modification to be performed

**Returns** This `MaterialToMaterialSingleEquilibriumCalculation` object

---

**class** `tc_python.material_to_material.MaterialToMaterialSingleEquilibriumResult`

**Bases:** `tc_python.single_equilibrium.SingleEquilibriumResult`

Result of a Material To Material calculation for a single fraction of material B, it can be evaluated using a quantity or Console Mode syntax.

### change_pressure

Change the pressure and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with higher performance. The properties are calculated at the new pressure using the phase amount, temperature and composition of phases from the initial equilibrium. Use `get_value_of()` to obtain them.

**Parameters** pressure – The pressure [Pa]

**Returns** This `MaterialToMaterialSingleEquilibriumResult` object
**change_temperature** *(temperature: float)*

Change the temperature and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with high performance. The properties are calculated at the new temperature using the phase amount, pressure and composition of phases from the initial equilibrium. Use `get_value_of()` to obtain them.

**Note:** This is typically used when calculating room temperature properties (e.g. density) for a material when it is assumed that the equilibrium phase amount and composition freeze-in at a higher temperature during cooling.

**Parameters** `temperature` – The temperature [K]

**Returns** This `MaterialToMaterialSingleEquilibriumResult` object

`get_components()` → List[str]

Returns the names of the components selected in the system (including any components auto-selected by the database(s)).

**Returns** The names of the selected components

`get_conditions()` → List[str]

Returns the conditions.

**Returns** The selected conditions

`get_phases()` → List[str]

Returns the phases present in the system due to its configuration. It also contains all phases that have been automatically added during the calculation, this is the difference to the method `System.get_phases_in_system()`.

**Returns** The names of the phases in the system including automatically added phases

`get_stable_phases()` → List[str]

Returns the stable phases (i.e. the phases present in the current equilibrium).

**Returns** The names of the stable phases

`get_value_of` *(quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str]) → float*  

Returns a value from a single equilibrium calculation.

**Parameters** `quantity` – The thermodynamic quantity to get the value of; a Console Mode syntax strings can be used as an alternative (for example “NPM(FCC_A1)”)  

**Returns** The requested value

`run_poly_command` *(command: str)*

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine. This affects only the state of the result object.

**Parameters** `command` – The Thermo-Calc Console Mode command

**Returns** This `MaterialToMaterialSingleEquilibriumResult` object

**Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.
**Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

```python
save_to_disk(path: str)
```

Saves the result to disk. Note that the result is a folder, containing potentially many files. The result can later be loaded with `load_result_from_disk()`

- **Parameters** `path` – the path to the folder you want the result to be saved in. It can be relative or absolute.
- **Returns** this `MaterialToMaterialSingleEquilibriumResult` object

```python
class tc_python.material_to_material.TemperatureAxis(from_temperature: float = 1000, to_temperature: float = 3000, start_temperature: float = 2000)
```

A temperature calculation axis.

```python
class tc_python.material_to_material.TemperatureCondition(temperature: float = 1000.0)
```

A constant temperature condition.

### 5.1.9 Package “process_metallurgy”

#### 5.1.9.1 Module “base”

```python
class tc_python.process_metallurgy.base.AbstractAddition
```

The base class for representing an addition to an equilibrium calculation or process simulation.

```python
def get_composition() -> Dict[str, float]
```

Returns the composition of the addition - without containing a dependent component.

- **Returns** The composition [in the unit provided by `getCompositionUnit()`]

```python
abstract def get_composition_unit()
```

Returns the composition unit used in this addition.

- **Returns** The composition unit

```python
def get_dependent_component() -> str
```

Returns the dependent component.

- **Returns** The dependent component or an empty string if no dependent component is defined

```python
def get_elements() -> Set[str]
```

Returns all elements of the addition.

- **Returns** The elements

```python
def get_id() -> str
```

Returns the unique ID of the addition.

- **Returns** The unique ID of the addition
get_temperature() → float
Returns the temperature of the addition. This refers to the temperature before it is added to the process.

   Returns The temperature [K]

is_do_scale() → bool
Returns if the composition of the addition is being scaled to 100% / 1 or not.

   Returns If the composition is scaled

is_empty() → bool
Returns if the addition is “empty”, i.e., has zero amount.

   Returns If the addition is empty

class tc_python.process_metallurgy.base.ActivityReference(value)
   Bases: enum.Enum

The reference for a slag activity calculation. The actual reference phase depends on the component for which the activity is request and can be obtained by using these methods on the result object: tc_python.process_metallurgy.process.ProcessSimulationResult.get_formula_for_activity_of_slag() or tc_python.process_metallurgy.equilibrium.EquilibriumResult.get_formula_for_activity_of_slag().

LIQUID = 1
   The reference is liquid slag.

SOLID = 0
   The reference is solid slag.

class tc_python.process_metallurgy.base.PhaseGroup(value)
   Bases: enum.Enum

The phase group, such a group is collecting all phases that belong to a certain type.

ALL_METAL = 2
   All metal phases.

ALL_SLAG = 5
   All slag phases.

GAS = 6
   All gas phases.

LIQUID_METAL = 0
   All liquid metal phases.

LIQUID_SLAG = 4
   All liquid slag phases.

SOLID_METAL = 1
   All solid metal phases.

SOLID_SLAG = 3
   All solid slag phases.

class tc_python.process_metallurgy.base.ProcessDatabase(value)
   Bases: enum.Enum

The database used for a Process Metallurgy calculation.

LATEST = 0
   The latest database available.
**OXDEMO = 1**
The database OXDEMO.

**TCOX10 = 4**
The database TCOX10.

**TCOX11 = 5**
The database TCOX11.

**TCOX8 = 2**
The database TCOX8.

**TCOX9 = 3**
The database TCOX9.

```python
class tc_python.process_metallurgy.base.ProcessMetallurgyOptions
    Bases: tc_python.single_equilibrium.SingleEquilibriumOptions

The options for a process metallurgy calculation.

disable_approximate_driving_force_for_metastable_phases()
    Disables the approximation of the driving force for metastable phases.

    **Default:** Enabled

    **Note:** When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

    If it is important that these driving forces are correct, use disable_approximate_driving_force_for_metastable_phases() to force the calculation to converge for the metastable phases.

Returns This ProcessMetallurgyOptions object
```

```python
disable_control_step_size_during_minimization()
    Disables step size control during minimization (non-global).

    **Default:** Enabled

    **Returns** This ProcessMetallurgyOptions object
```

```python
disable_force_positive_definite_phase_hessian()
    Disables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for “Hessian minimization”.

    **Default:** Enabled

    **Returns** This ProcessMetallurgyOptions object
```

```python
enable_approximate_driving_force_for_metastable_phases()
    Enables the approximation of the driving force for metastable phases.

    **Default:** Enabled
```

**Note:** When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.
If it is important that these driving forces are correct, use `disable_approximate_driving_force_for_metastable_phases()` to force the calculation to converge for the metastable phases.

**Returns** This `ProcessMetallurgyOptions` object

`enable_control_step_size_during_minimization()` Enables step size control during normal minimization (non-global).

**Default:** Enabled

**Returns** This `ProcessMetallurgyOptions` object

`enable_force_positive_definite_phase_hessian()` Enables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for “Hessian minimization”.

**Default:** Enabled

**Returns** This `ProcessMetallurgyOptions` object

`set_global_minimization_max_grid_points`(max_grid_points: int = 2000) Sets the maximum number of grid points in global minimization. **Only applicable if global minimization is actually used.**

**Default:** 2000 points

**Parameters** max_grid_points – The maximum number of grid points

**Returns** This `ProcessMetallurgyOptions` object

`set_max_no_of_iterations`(max_no_of_iterations: int = 2000) Sets the maximum number of iterations for the CALPHAD minimizer.

**Default:** max. 2000 iterations

**Note:** As some models give computation times of more than 1 CPU second/iteration, this number is also used to check the CPU time and the calculation stops if 500 CPU seconds/iterations are used.

**Parameters** max_no_of_iterations – The max. number of iterations

**Returns** This `ProcessMetallurgyOptions` object

`set_process_minimization_policy`(minimization_policy: tc_python.process_metallurgy.base.ProcessMinimizationPolicy) Sets the minimization policy for the process metallurgy calculation. It is possible to choose policies that try different methods if one method fails.

**Parameters** minimization_policy – The minimization policy to be used

**Returns** This `ProcessMetallurgyOptions` object

`set_required_accuracy`(accuracy: float = 1e-06) Sets the required relative accuracy.

**Default:** 1.0E-6

**Note:** This is a relative accuracy, and the program requires that the relative difference in each variable must be lower than this value before it has converged. A larger value normally means fewer iterations.
but less accurate solutions. The value should be at least one order of magnitude larger than the machine

**Parameters** **accuracy** – The required relative accuracy

**Returns** This `ProcessMetallurgyOptions` object

`set_smallest_fraction` *(smallest_fraction: float = 1e-16)*

Sets the smallest fraction for constituents that are unstable.

It is normally only in the gas phase that you can find such low fractions.

The **default value** for the smallest site-fractions is 1E-16 for all phases except for IDEAL phase with one

sublattice site (such as the GAS mixture phase in many databases) for which the default value is always as

1E-30.

**Parameters** **smallest_fraction** – The smallest fraction for constituents that are unstable

**Returns** This `ProcessMetallurgyOptions` object

```
class tc_python.process_metallurgy.base.ProcessMinimizationPolicy(value)
Bases: enum.Enum
```

The policy for the CALPHAD minimization routine used in a calculation.

**Note:** This affects the runtime and stability of a calculation. Global minimization is more stable but more time-

consuming. Local minimization is much faster but can miss new phases coming up. Global test is a compromise

between both approaches.

```
GLOBAL = 0
    Always use global minimization.

GLOBAL_TEST = 1
    Always use global test.

GLOBAL_TEST__GLOBAL = 5
    First try global test, if that fails use global minimization.

LOCAL = 2
    Always use local minimization.

LOCAL__GLOBAL_TEST = 3
    First try local minimization, if that fails use global test.

LOCAL__GLOBAL_TEST__GLOBAL = 4
    First try local minimization, if that fails try global test and if that fails use global minimization.
```

```
class tc_python.process_metallurgy.base.SlagProperty(value)
Bases: enum.Enum
```

The slag property, different definitions are available. The actual definition of a certain

slag property for the current system can be obtained using these methods on the result ob-

ject:


getFormulaForSlagProperty() or tc_python.process_metallurgy.equilibrium.

EquilibriumResult.getFormulaForSlagProperty().

---

5.1. Calculations 187
Note: If not all components required by the definition of slag property are available in a given system, the slag property will return NaN.

B2 = 0
Basicity B2 (based on: CaO / SiO2).

B3 = 1
Basicity B3 (based on: CaO / MgO / SiO2).

B4 = 3
Basicity B4 (based on: CaO / MgO / SiO2 / Al2O3).

BAS2 = 2
Basicity Bas2 (based on: CaO / MgO / SiO2 / Al2O3 / TiO2).

BELLS_RATIO = 4
Basicity Bell’s ratio (based on: CaO / MgO / SiO2 / Al2O3).

LOG_10_SULPHUR_CAPACITY = 5
Logarithmic sulphur capacity of the slag.

LS = 6
Sulphur capacity Ls of the slag.

class tc_python.process_metallurgy.base.SlagType(value)
Bases: enum.Enum

The type of slag considered for a slag property calculation.

ALL = 2
The slag property is calculated for all slag, i.e. for both the liquid and solid slag.

LIQUID = 1
The slag property is calculated for all liquid slag.

SOLID = 0
The slag property is calculated for all solid slag.

5.1.9.2 Module “equilibrium”

class tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition
Bases: tc_python.process_metallurgy.base.AbstractAddition

The base class for representing an addition to an equilibrium calculation.

set_amount (amount: float)
Change the amount of the addition.

Parameters

amount – The new amount [in the amount unit of this addition]

Returns

This AbstractEquilibriumAddition object

set_component_composition (component: str, content: float)
Change the composition of a component of the addition.

Parameters

• component – The component to be changed

• content – The new content of the component [in the composition unit defined for this addition]
class tc_python.process_metallurgy.equilibrium.AdiabaticEquilibriumCalculation(calculator):

Bases: tc_python.process_metallurgy.equilibrium.EquilibriumCalculation

An adiabatic Process Metallurgy equilibrium calculation. Such calculations can for example be used to determine the global equilibrium state of a process.

add_addition (addition: tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition)

Add an addition to the calculation.

Parameters addition – A EquilibriumAddition or EquilibriumGasAddition

Returns This AdiabaticEquilibriumCalculation object

add_poly_command (command: str)

Adds a Thermo-Calc Console syntax POLY module command which will be executed when performing the calculation using the calculate() method.

If multiple commands are added, they will be executed in the order of addition. Each command will only be executed one.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

Parameters command – The POLY module command in Thermo-Calc console syntax

Returns This AdiabaticEquilibriumCalculation object

calculate (timeout_in_minutes: float = 0.0) → tc_python.process_metallurgy.equilibrium.EquilibriumResult

Runs the Process Metallurgy equilibrium calculation.

Parameters timeout_in_minutes – The calculation will be aborted after that time, default: no timeout

Returns A new EquilibriumResult object

remove_addition (addition: tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition)

Removes an addition from the calculation.

Parameters addition – The addition to be removed

Returns This AdiabaticEquilibriumCalculation object

remove_all_additions ()

Removes all additions from the calculation.

Returns This AdiabaticEquilibriumCalculation object

set_pressure (pressure: float = 100000.0)

Sets the pressure.

Parameters pressure – The pressure [Pa]

Returns This AdiabaticEquilibriumCalculation object

5.1. Calculations
**with_options** *(options: tc_python.process_metallurgy.base.ProcessMetallurgyOptions)*

Sets the options for the calculation.

**Parameters**
- **options** – The options

**Returns** This *AdiabaticEquilibriumCalculation* object

---

**class**  
*tc_python.process_metallurgy.equilibrium.EquilibriumAddition*(composition: Dict[str, float], amount: float, temperature: float = 293.15, composition_unit: tc_python.utils.CompositionUnit = CompositionUnit.MASS_PERCENT: 0, do_scale: bool = False)

**Bases:**  
*tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition*

An addition to an equilibrium calculation.

**Tip:** By setting 

 shortest  

 **do_scale=True**, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not summing to 100% / 1. An example could be a slag addition which is provided like this: 90 wt-% CaO - 5 wt-% Al2O3 - 4 wt-% SiO2.

---

**Parameters**
- **composition** – The composition
- **amount** – The amount [kg]
- **temperature** – The initial addition temperature *(default: 20 °C)* [K]
- **composition_unit** – The composition unit
- **do_scale** – If the composition is scaled to 100% / fraction of 1

**get_amount** () → float

Returns the amount of this addition.

**Returns** The amount [kg]

**get_composition_unit** () → tc_python.utils.CompositionUnit

Returns the composition unit used in this addition.

**Returns** The composition unit

---

**class**  
*tc_python.process_metallurgy.equilibrium.EquilibriumCalculation*(metallurgical_reaction)

**Bases:**  
*object*

A Process Metallurgy equilibrium calculation. Such calculations can for example be used to determine the global equilibrium state of a process.
abstract add_addition

(addition: tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition)

Adds an addition to the calculation.

Parameters

addition – The addition

Returns

This EquilibriumCalculation object

abstract add_poly_command

(command: str)

Adds a Thermo-Calc Console syntax POLY module command which will be executed when performing the calculation using the calculate() method.

If multiple commands are added, they will be executed in the order of addition. Each command will only be executed one.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Parameters

command – The POLY module command in Thermo-Calc console syntax

Returns

This EquilibriumCalculation object

abstract calculate

(timeout_in_minutes: float = 0.0)  \rightarrow
tc_python.process_metallurgy.equilibrium.EquilibriumResult

Runs the Process Metallurgy equilibrium calculation.

Parameters

timeout_in_minutes – The calculation will be aborted after that time, default: no timeout

Returns

A new EquilibriumResult object

abstract remove_addition

(addition: tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition)

Removes an addition from the calculation.

Parameters

addition – The addition to be removed

Returns

This EquilibriumCalculation object

abstract remove_all_additions

Removes all additions from the calculation.

Returns

This EquilibriumCalculation object

abstract set_pressure

(pressure: float = 100000.0)

Sets the pressure.

Parameters

pressure – The pressure [Pa]

Returns

This EquilibriumCalculation object

update_addition

(addition: tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition)

Replaces an already added addition with an updated one. This is usually used to change the composition or amount of an addition while iterating over them. Typically, this is done for stepping or mapping calculations.

Note: The calculation must already contain the addition object to be updated.
Parameters `addition` – The new addition containing updated values

Returns This IsoThermalMetallurgyCalculation object

**abstract with_options** (options: tc_python.process_metallurgy.base.ProcessMetallurgyOptions)

Sets the options for the calculation.

Parameters `options` – The options

Returns This EquilibriumCalculation object

```python
class tc_python.process_metallurgy.equilibrium.EquilibriumGasAddition (composition: Dict[str, float], amount: float, temperature: float = 293.15, amount_unit: tc_python.utils.GasAmountUnit = <GasAmountUnit.NORM_CUBIC_METER: 0>, composition_unit: tc_python.utils.GasCompositionUnit = <GasCompositionUnit.VOLUME_PERCENT: 4>, do_scale: bool = False)
```

Bases: tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition

A gas addition to an equilibrium calculation.

Tip: By setting `do_scale=True`, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not summing to 100% / 1. An example could be a gas addition which is provided like this: 90 vol-% Ar - 10 vol-% O2.

```python
def get_amount () -> float
    Returns the amount of this addition.
```

192 Chapter 5. API Reference
Note: The amount unit can be obtained using `get_amount_unit()`.

Returns The amount [in the amount unit]

`get_amount_unit() \rightarrow tc_python.utils.GasAmountUnit`
Returns the amount unit used in this addition.

Returns The amount unit

`get_composition_unit() \rightarrow tc_python.utils.GasCompositionUnit`
Returns the composition unit used in this addition.

Returns The composition unit

```python
class tc_python.process_metallurgy.equilibrium.EquilibriumResult(result)
Bases: tc_python.abstract_base.AbstractResult
```
The result of a Process Metallurgy equilibrium calculation.

Returns the activity of a component in the slag.

Parameters
- `component` – The component
- `reference` – The reference for the activity, can be liquid or solid slag, default: liquid

Returns The activity of the component [-]

`get_amount() \rightarrow float`
Returns the total amount.

Returns The total amount [kg]

`get_amount_of_elements() \rightarrow Dict[str, float]`
Returns the amount of each element.

Returns The amount of the elements [kg]

`get_amount_of_phase_groups() \rightarrow Dict[tc_python.process_metallurgy.base.PhaseGroup, float]`
Returns the amount of each phase group (e.g., for example all liquid slag).

Returns The amount of the phase groups [kg]

`get_amount_of_phases() \rightarrow Dict[str, float]`
Returns the amount of each phase.

Returns The amount of the phases [kg]

`get_components() \rightarrow Set[str]`
Returns all components defined for the elements present in this result.

Returns The components present in this result

`get_composition(composition_unit: tc_python.utils.CompositionUnit = <CompositionUnit.MASS_PERCENT: 0>) \rightarrow Dict[str, float]`
Returns the composition of the result.

Parameters `composition_unit` – The composition unit, default: mass percent

Returns The composition
get_composition_of_phase (phase: str, composition_unit: tc_python.utils.CompositionUnit = <CompositionUnit.MASS_PERCENT: 0>, composition_type: tc_python.utils.CompositionType = <CompositionType.COMPONENT: 1>) → Dict[str, float]

Returns the composition of a phase in the result.

Parameters

• **phase** – The phase name

• **composition_unit** – The composition unit, default: mass percent

• **composition_type** – Defines if the composition is given by element (e.g., 75 wt-% Fe - 25 wt-% Cr) or by component (e.g. 65 wt-% Al2O3 - 35 wt-% CaO). In case of a metallic phase, the composition is given by element even if component is selected. Default: by component.

Returns The composition

get_composition_of_phase_group (phase_group: tc_python.process_metallurgy.base.PhaseGroup, composition_unit: tc_python.utils.CompositionUnit = <CompositionUnit.MASS_PERCENT: 0>, composition_type: tc_python.utils.CompositionType = <CompositionType.COMPONENT: 1>) → Dict[str, float]

Returns the composition of a phase group (e.g., all liquid slag) in the result.

Parameters

• **phase_group** – The phase group

• **composition_unit** – The composition unit, default: mass percent

• **composition_type** – Defines if the composition is given by element (e.g., 75 wt-% Fe - 25 wt-% Cr) or by component (e.g. 65 wt-% Al2O3 - 35 wt-% CaO). In case of a metallic phase, the composition is given by element even if component is selected. Default: by component.

Returns The composition

get_elements () → Set[str]

Returns all elements defined for the result.

Returns All elements present in this result


Returns the Thermo-Calc Console syntax formula used for calculating the activity of a component in the slag (e.g. AC(AL2O3, IONIC_LIQ). The actual activity can be obtained using get_activity_of_slag().

Parameters

• **component** – The component

• **reference** – The reference for the activity, can be liquid or solid slag, default: liquid slag

Returns The formula for calculating the activity

get_formula_for_slag_property (slag_property: tc_python.process_metallurgy.base.SlagProperty, slag_type: tc_python.process_metallurgy.base.SlagType = <SlagType.ALL: 2>) → str

Returns the Thermo-Calc Console syntax formula used for calculating a property of the slag (e.g.
The actual slag property can be obtained using `get_slag_property()`. The slag property can be calculated for different parts of the slag:

**Parameters**

- `slag_property` – The slag property
- `slag_type` – The part of the slag for which the property will be calculated. Can be all slag, the liquid or the solid slag. Default: all slag

**Returns** The formula for calculating the slag property

```python
get_fraction_of_phase_groups(unit: tc_python.utils.PhaseUnit = <PhaseUnit.MASS_FRACTION: 1>) → Dict[tc_python.process_metallurgy.base.PhaseGroup, float]
```

Returns the fraction of the phase groups (e.g., all liquid slag) in the result.

**Parameters** `unit` – The unit of the fraction, default: volume fraction

**Returns** The phase fractions

```python
get_fraction_of_phases(unit: tc_python.utils.PhaseUnit = <PhaseUnit.MASS_FRACTION: 1>) → Dict[str, float]
```

Returns the fraction of the stable phases in the result.

**Parameters** `unit` – The unit of the fraction, default: volume fraction

**Returns** The phase fractions

```python
get_gas_components() → Set[str]
```

Returns all components of the gas phase defined for the elements present in this result.

**Returns** The components of the gas phase present in this result

```python
get_oxygen_partial_pressure() → float
```

Returns the partial pressure of oxygen in the result.

**Returns** The partial pressure [Pa]

```python
get_pressure() → float
```

Returns the pressure in the result.

**Returns** The pressure [Pa]

```python
get_slag_property(slag_property: tc_python.process_metallurgy.base.SlagProperty, slag_type: tc_python.process_metallurgy.base.SlagType = <SlagType.ALL: 2>) → float
```

Returns a property of the slag. These properties are mostly used to describe the property of a slag to pick up sulfur.

**Parameters**

- `slag_property` – The slag property
- `slag_type` – The part of the slag for which the property will be calculated. Can be all slag, the liquid or the solid slag. Default: all slag

**Returns** The slag property [unit depending on the property]
Returns

The stable phases

get_temperature() → float

Returns the temperature in the result.

Returns

The temperature [K]

get_value_of(classic_expression: str) → float

Returns a value for a thermodynamic quantity.

Warning: It should normally not be required to use this method, use the appropriate method available in the API instead.

Parameters

classic_expression

– The thermodynamic quantity to get the value of in Thermo-Calc Console Mode syntax (for example “NPM(FCC_A1)”)

Returns

The requested value

get_viscosity_dynamic_of_phase(phase: str) → float

Returns the dynamic viscosity of a phase in the result.

Parameters

phase

– The phase name

Returns

The dynamic viscosity [Pa*s]

get_viscosity_kinematic_of_phase(phase: str) → float

Returns the kinematic viscosity of a phase in the result.

Parameters

phase

– The phase name

Returns

The kinematic viscosity [m**2/s]

class tc_python.process_metallurgy.equilibrium.IsoThermalEquilibriumCalculation(calculation)

Bases: tc_python.process_metallurgy.equilibrium.EquilibriumCalculation

An isothermal Process Metallurgy equilibrium calculation. Such calculations can for example be used to determine the global equilibrium state of a process.

add_addition(addition: tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition)

Adds an addition to the calculation.

Parameters

addition

– A EquilibriumAddition or EquilibriumGasAddition

Returns

This IsoThermalEquilibriumCalculation object

add_poly_command(command: str)

Adds a Thermo-Calc Console syntax POLY module command which will be executed when performing the calculation using the calculate() method.

If multiple commands are added, they will be executed in the order of addition. Each command will only be executed one.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).
Parameters `command` – The POLY module command in Thermo-Calc console syntax

Returns This `IsoThermalEquilibriumCalculation` object

`calculate` *(timeout_in_minutes: float = 0.0) → tc_python.process_metallurgy.equilibrium.EquilibriumResult*

Runs the Process Metallurgy equilibrium calculation.

Parameters `timeout_in_minutes` – The calculation will be aborted after that time, default: no timeout

Returns A new `EquilibriumResult` object

`remove_addition` *(addition: tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition)*

Removes an addition from the calculation.

Parameters `addition` – The addition to be removed

Returns This `IsoThermalEquilibriumCalculation` object

`remove_all_additions` *

Removes all additions from the calculation.

Returns This `IsoThermalEquilibriumCalculation` object

`set_pressure` *(pressure: float = 100000.0)*

Sets the pressure.

Parameters `pressure` – The pressure [Pa]

Returns This `IsoThermalEquilibriumCalculation` object

`set_temperature` *(temperature: float)*

Sets the temperature.

Parameters `temperature` – The temperature [K]

Returns This `IsoThermalEquilibriumCalculation` object

`update_addition` *(addition: tc_python.process_metallurgy.equilibrium.AbstractEquilibriumAddition)*

Replaces an already added addition with an updated one.

Tip: This is usually used to change the composition or amount of an addition while iterating over multiple values. Typically, this is done for stepping or mapping calculations.

Note: The calculation must already contain the addition object to be updated.

Parameters `addition` – A previously added addition object with the updated values

Returns This `IsoThermalEquilibriumCalculation` object

`with_options` *(options: tc_python.process_metallurgy.base.ProcessMetallurgyOptions)*

Sets the options for the calculation.

Parameters `options` – The options

Returns This `IsoThermalEquilibriumCalculation` object

5.1. Calculations
5.1.9.3 Module “process”

class tc_python.process_metallurgy.process.AbstractContinuousAddition
   Bases: tc_python.process_metallurgy.process.AbstractProcessAddition

   The base class representing an addition in a process simulation that is added continuously over a period of time.

class tc_python.process_metallurgy.process.AbstractProcessAddition
   Bases: tc_python.process_metallurgy.base.AbstractAddition

   The base class for representing an addition in a process simulation.

class tc_python.process_metallurgy.process.AbstractSingleTimeAddition
   Bases: tc_python.process_metallurgy.process.AbstractProcessAddition

   The base class representing an addition in a process simulation that is added at a distinct time point.

class tc_python.process_metallurgy.process.BulkZone
   Parameters:
   • density: float,
   • phase_group_to_transfer: tc_python.process_metallurgy.base.PhaseGroup,
   • name: str
   Bases: tc_python.process_metallurgy.process.Zone

   A bulk zone in a process simulation, this is representing a large volume in the process, for example the steel melt or the top slag. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

   Tip: This is a generic class and seldom used directly. Use instead MetalBulkZone or SlagBulkZone.

   add_addition (addition: tc_python.process_metallurgy.process.AbstractSingleTimeAddition, time: float = 0.0)
   Adds a single-time addition at the specified time point to the zone. The addition will be dissolved immediately.

   Parameters
   • addition – A SingleTimeAddition or SingleTimeGasAddition
   • time – The time point [s]

   Returns This BulkZone object

   add_continuous_addition (addition: tc_python.process_metallurgy.process.AbstractContinuousAddition, from_time: float = 0.0, to_time: float = nan)
   Adds a constant addition continuously during the specified time period to the zone. All added material will be dissolved immediately.

   Parameters
   • addition – A ContinuousAddition or ContinuousGasAddition
   • from_time – The start time point [s]
   • to_time – The end time point [s]

   Returns This BulkZone object

   add_power (power: float, from_time: float = 0.0, to_time: float = nan)
   Adds a constant power during a specified time period to the zone (for example heating or cooling).

   Parameters
   • power – The power [MW]
• **from_time** – The start time point [s]

• **to_time** – The end time point [s]

**Returns** This BulkZone object

**disable_degassing()**
Disables degassing for this zone, i.e. all gas formed at any time step will be staying in this zone.

**Returns** This BulkZone object

**enable_degassing()**
Enables degassing for this zone, i.e. any gas formed at any time step will be removed after that time step. This gas will be transferred into the exhaust gas zone. **This is the default.**

**Returns** This BulkZone object

**get_density() → float**
Returns the density of the zone

**Returns** The density [kg/m**3**]

**get_elements() → Set[str]**
Returns the elements present in the zone. The elements are determined by the additions.

**Returns** The elements

**get_id() → str**
Returns the unique id of the zone. :return: The zone id

**get_phase_group_to_transfer() → tc_python.process_metallurgy.base.PhaseGroup**
Returns the phase group that is transferred from the attached reaction zones back to this zone after each time step.

**Returns** The phase group

**is_degassing_enabled() → bool**
Returns if degassing is enabled in the zone.

**Returns** If degassing is enabled

---

**class tc_python.process_metallurgy.process.ContinuousAddition**(composition: Dict[str, float], rate: float, temperature: float = 293.15, composition_unit: tc_python.utils.CompositionUnit = <CompositionUnit.MASS_PERCENT: 0>, do_scale: bool = False)  

**Bases**: tc_python.process_metallurgy.process.AbstractContinuousAddition

An addition in a process simulation that is added continuously during a period of time.

It is assumed that the material added during that period is dissolved instantaneously.

**Tip:** By setting `do_scale=True`, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not scaling to 100% / 1. An example could be a slag addition which is provided like this: 90 wt-% CaO - 5 wt-% Al2O3 - 4 wt-% SiO2.

---

5.1. Calculations
Parameters

- **composition** – The composition
- **rate** – The rate of addition [kg/s]
- **temperature** – The initial addition temperature (default: 20 °C) [K]
- **composition_unit** – The composition unit
- **do_scale** – If the composition is scaled to 100% / fraction of 1

`get_composition_unit()` → `tc_python.utils.CompositionUnit`

Returns the composition unit used in this addition.

**Returns** The composition unit

`get_rate()` → float

Returns the rate of addition.

**Returns** The addition rate [kg/s]

```python
class tc_python.process_metallurgy.process.ContinuousGasAddition(
    composition: Dict[str, float], rate: float, temper-
    ature: float = 293.15, rate_unit: tc_python.utils.GasRateUnit
    = <Gas-
    RateU-
    nit.NORM_CUBIC_METER_PER_SEC: 0>, compo-
    sition_unit:
    tc_python.utils.GasCompositionUnit
    = <Gas-
    Compositio-
    nUnit.VOLUME_PERCENT: 4>,
    do_scale: bool = False)
```

**Bases:** `tc_python.process_metallurgy.process.AbstractContinuousAddition`

A gas addition in a process simulation that is added continuously during a period of time.

It is assumed that the gas added during that period is dissolved instantaneously.

**Tip:** By setting `do_scale=True`, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not scaling to 100% / 1. An example could be a gas addition which is provided like this: 90 vol-% Ar - 10 vol-% O2.

`get_composition_unit()` → `tc_python.utils.GasCompositionUnit`

Returns the composition unit used in this addition.

**Returns** The composition unit

`get_rate()` → float

Returns the rate of addition.
Note: The rate unit can be obtained using `get_rate_unit()`.

**Returns** The addition rate [in the rate unit]

`get_rate_unit()` → `tc_python.utils.GasRateUnit`

Returns the rate unit used in this addition.

**Returns** The rate unit

class `tc_python.process_metallurgy.process.ExhaustGasResult` (result)

Bases: object

A result representing the exhaust gas zone, here all exhaust gas generated during the process is accumulated.

The data is returned for each time point of the process simulation. These time points can be obtained from this method: `ProcessSimulationResult.get_time_points()`.

`get_amount()` → List[float]

Returns the amount of exhaust gas present at each time point.

This is the amount of gas accumulated since the beginning of the process.

**Returns** The accumulated amount of gas at each time point [kg]

`get_amount_of_components()` → Dict[str, List[float]]

Returns the amount of each exhaust gas component present at each time point.

This is the amount of gas accumulated since the beginning of the process. This is different from the current composition at each time point obtained using `get_composition()`.

**Returns** The accumulated amount of each gas component at each time point [kg]

`get_composition`(composition_type: `tc_python.utils.CompositionType` = <CompositionType.COMPONENT: 1>, unit: `tc_python.utils.CompositionUnit` = <CompositionUnit.MASS_PERCENT: 0>) → Dict[str, List[float]]

Returns the current composition of the exhaust gas zone at each time point. This is the composition at each time point. This is different from the accumulated amount obtained using `get_amount_of_components()`.

**Parameters**

- **composition_type** – The type of the composition, can be by gas component or by element, default: by gas component
- **unit** – The composition unit, default: mass percent

**Returns** The current composition of the gas components at each time point

`get_pressure()` → List[float]

Returns the pressure of the exhaust gas zone at each time point.

**Returns** The pressure [Pa]

`get_stable_phases()` → Set[str]

Returns the stable phases within the exhaust gas zone at each time point.

**Returns** The stable phases

`get_temperature()` → List[float]

Returns the temperature of the exhaust gas at each time point.

**Returns** The temperature at each time point [K]
\textbf{class} \texttt{tc\_python.process\_metallurgy.process.MassTransferCoefficients}

\begin{verbatim}
Bases: object

The mass transfer coefficients between a reaction zone and a bulk zone vs. time.

\textbf{add} (\textit{mass\_transfer\_coefficient: float, time: float = 0.0})

Adds the mass transfer coefficient valid beginning at a time point.

This value is valid until another value is defined for a later time point.

\textbf{Parameters}

\begin{itemize}
  \item \textit{mass\_transfer\_coefficient} – The mass transfer coefficient [m/s]
  \item \textit{time} – The time-point where the mass transfer coefficient begins to be valid [s]
\end{itemize}

\textbf{Returns} This \texttt{MassTransferCoefficients} object
\end{verbatim}

\textbf{class} \texttt{tc\_python.process\_metallurgy.process.MetalBulkZone} (\textit{density: float})

\begin{verbatim}
Bases: tc\_python.process\_metallurgy.process.Zone

A metallic bulk zone in a process simulation.

This is representing a large volume in the process, for example the steel melt. A zone is a volume in a process
that has identical temperature and composition. It has well-defined boundaries to other zones.

The name of this zone is automatically defined and unique.

\textbf{add\_addition} (\textit{addition: tc\_python.process\_metallurgy.process.AbstractSingleTimeAddition, time: float = 0.0})

Adds a single-time addition at the specified time point to the zone. The addition will be dissolved immedi-
ately.

\textbf{Parameters}

\begin{itemize}
  \item \textit{addition} – A \texttt{SingleTimeAddition} or \texttt{SingleTimeGasAddition}
  \item \textit{time} – The time point [s]
\end{itemize}

\textbf{Returns} This \texttt{MetalBulkZone} object

\textbf{add\_continuous\_addition} (\textit{addition: tc\_python.process\_metallurgy.process.AbstractContinuousAddition, from\_time: float = 0.0, to\_time: float = nan})

Adds a constant addition continuously during the specified time period to the zone. All added material will
be dissolved immediately.

\textbf{Parameters}

\begin{itemize}
  \item \textit{addition} – A \texttt{ContinuousAddition} or \texttt{ContinuousGasAddition}
  \item \textit{from\_time} – The start time point [s]
  \item \textit{to\_time} – The end time point [s]
\end{itemize}

\textbf{Returns} This \texttt{MetalBulkZone} object

\textbf{add\_power} (\textit{power: float, from\_time: float = 0.0, to\_time: float = nan})

Adds a constant power during a specified time period to the zone (for example heating or cooling).

\textbf{Parameters}

\begin{itemize}
  \item \textit{power} – The power [MW]
  \item \textit{from\_time} – The start time point [s]
  \item \textit{to\_time} – The end time point [s]
\end{itemize}

\textbf{Returns} This \texttt{MetalBulkZone} object
disable_degassing()
Disables degassing for this zone, i.e. all gas formed at any time step will be staying in this zone.

Returns This MetalBulkZone object

enable_degassing()
Enables degassing for this zone, i.e. any gas formed at any time step will be removed after that time step. This gas will be transferred into the exhaust gas zone. This is the default.

Returns This MetalBulkZone object

get_density() \rightarrow float
Returns the density of the zone

Returns The density [kg/m**3]

get_elements() \rightarrow Set[str]
Returns the elements present in the zone. The elements are determined by the additions.

Returns The elements

get_id() \rightarrow str
Returns the unique name / id of the zone.

Returns The zone name / id

get_phase_group_to_transfer() \rightarrow tc_python.process_metallurgy.base.PhaseGroup
Returns the phase group that is transferred from the attached reaction zones back to this zone after each time step.

Returns The phase group

is_degassing_enabled() \rightarrow bool
Returns if degassing is enabled

Returns If degassing is enabled

class tc_python.process_metallurgy.process.ProcessSimulationCalculation(calculation)
Bases: object
A Process Metallurgy process simulation. Such calculations represent complete metallurgical processes with several zones and simulate their evolution over time.

calculate (timeout_in_minutes: float = 0.0) \rightarrow tc_python.process_metallurgy.process.ProcessSimulationResult
Runs the Process Metallurgy process simulation.

Parameters timeout_in_minutes – The calculation will be aborted after that time, default: no timeout

Returns A new ProcessSimulationResult object

set_end_time (end_time: float)
Sets the end time of a process.

Parameters end_time – The end time point [s]

Returns This ProcessSimulationCalculation object

set_initial_time_step (initial_time_step: float = 1.0)
Sets the initial time step used in the process simulation.

Note: All later time steps are automatically determined to limit the expected temperature change during that step, this is controlled by set_max_allowed_temp_change_per_step().
Parameters `initial_time_step` – The initial time step [s]

Returns This `ProcessSimulationCalculation` object

`set_max_allowed_temp_change_per_step(max_allowed_temp_change: float = 10.0)`
The maximum allowed temperature change per time step. This is implicitly also limiting the composition change during a time step and required for numerical stability.

Parameters `max_allowed_temp_change` – The maximum allowed temperature change [K]

Returns This `ProcessSimulationCalculation` object

`set_max_time_step(max_time_step: float = 180.0)`
The maximum time step chosen by the automatic time step control.

Note: All time steps are automatically determined to limit the expected temperature change during that step, this is controlled by `set_max_allowed_temp_change_per_step()`.

Parameters `max_time_step` – The maximum time step [s]

Returns This `ProcessSimulationCalculation` object

`set_min_time_step(min_time_step: float = 1.0)`
The minimum time step chosen by the automatic time step control.

Note: All time steps are automatically determined to limit the expected temperature change during that step, this is controlled by `set_max_allowed_temp_change_per_step()`.

Parameters `min_time_step` – The minimum time step [s]

Returns This `ProcessSimulationCalculation` object

`set_pressure(pressure: float = 100000.0)`
Sets a constant pressure during the complete process.

Parameters `pressure` – The pressure [Pa]

Returns This `ProcessSimulationCalculation` object

`set_pressure_in_time_period(pressure_in_pa: float, from_time: float = 0.0, to_time: float = nan)`
Sets a constant pressure during a time period. Default: 1.0e5 Pa.

Parameters

• `pressure_in_pa` – The pressure [Pa]
• `from_time` – The start time [s]
• `to_time` – The end time [s]

Returns This `ProcessSimulationCalculation` object

`with_options(options: tc_python.process_metallurgy.base.ProcessMetallurgyOptions)`
Sets the options for the process simulation.

Parameters `options` – The options
with_reactio_m_zone (reaction_zone: tc_python.process_metallurgy.process.ReactionZone)
Sets the reaction zone of the process simulation. The bulk zones attached to this reaction zone are configured in the reaction zone object.

Note: In the present release, only one reaction zone is supported.

Parameters reaction_zone – The reaction zone object
Returns This ProcessSimulationCalculation object

class tc_python.process_metallurgy.process.ProcessSimulationResult (result)
Bases: tc_python.abstract_base.AbstractResult
The result of a Process Metallurgy process simulation.

Returns the activity of a component in the slag in a zone at each time point.

Parameters
• zone – The zone object or the zone name
• component – The component
• reference – The reference for the activity, can be liquid or solid slag, default: liquid slag

Returns The activity of the component at each time point [-]

get_amount (zone: Union[tc_python.process_metallurgy.process.Zone, str]) → List[float]
Returns the amount of a zone at each time point.

Parameters zone – The zone object or the zone name
Returns The amount at each time point [kg]

get_amount_of_elements () → Dict[str, List[float]]
Returns the total amount of each element in the simulation at each time point.

Returns The total amount of the elements at each time point [kg]

get_amount_of_phase_groups (zone: Union[tc_python.process_metallurgy.process.Zone, str]) → Dict[tc_python.process_metallurgy.base.PhaseGroup, List[float]]
Returns the amount of each phase group (e.g., for example all liquid slag) in a zone at each time point.

Parameters zone – The zone object or the zone name
Returns The amount of the phase groups at each time point [kg]

get_amount_of_phases (zone: Union[tc_python.process_metallurgy.process.Zone, str]) → Dict[str, List[float]]
Returns the amount of each phase in a zone at each time point.

Parameters zone – The zone object or the zone name
Returns The amount of the phases at each time point [kg]
get_components() → Set[str]
Returns all components defined in the simulation.

Returns The components

get_composition (zone: Union[tc_python.process_metallurgy.process.Zone, str], composition_unit: tc_python.utils.CompositionUnit = <CompositionUnit.MASS_PERCENT: 0>) → Dict[str, List[float]]
Returns the composition of a zone per element at each time point.

Parameters
- **zone** – The zone object or the zone name
- **composition_unit** – The composition unit, default: mass percent

Returns The composition at each time point

get_composition_of_phase (zone: Union[tc_python.process_metallurgy.process.Zone, str], phase: str, composition_unit: tc_python.utils.CompositionUnit = <CompositionUnit.MASS_PERCENT: 0>, composition_type: tc_python.utils.CompositionType = <CompositionType.COMPONENT: 1>) → Dict[str, List[float]]
Returns the composition of a phase in a zone at each time point.

Parameters
- **zone** – The zone object or the zone name
- **phase** – The phase name
- **composition_unit** – The composition unit, default: mass percent
- **composition_type** – Defines if the composition is given by element (e.g., 75 wt-% Fe - 25 wt-% Cr) or by component (e.g. 65 wt-% Al2O3 - 35 wt-% CaO). In case of a metallic phase, the composition is given by element even if component is selected. Default: by component.

Returns The composition at each time point

get_composition_of_phase_group (zone: Union[tc_python.process_metallurgy.process.Zone, str], phase_group: tc_python.process_metallurgy.base.PhaseGroup, composition_unit: tc_python.utils.CompositionUnit = <CompositionUnit.MASS_PERCENT: 0>, composition_type: tc_python.utils.CompositionType = <CompositionType.COMPONENT: 1>) → Dict[str, List[float]]
Returns the composition of a phase group (e.g., all liquid slag) in a zone at each time point.

Parameters
- **zone** – The zone object or the zone name
- **phase_group** – The phase group
- **composition_unit** – The composition unit, default: mass percent
- **composition_type** – Defines if the composition is given by element (e.g., 75 wt-% Fe - 25 wt-% Cr) or by component (e.g. 65 wt-% Al2O3 - 35 wt-% CaO). In case of a metallic phase, the composition is given by element even if component is selected. Default: by component.

Returns The composition at each time point

get_elements() → Set[str]
Returns all elements present in the simulation.
Returns The elements

get_enthalpy() → List[float]

Returns the total enthalpy of the process at each time point.

Returns The enthalpy at each time point [J]

get_exhaust_gas() → tc_python.process_metallurgy.process.ExhaustGasResult

Returns the result for the exhaust gas zone.

This result object can be used to evaluate the exhaust gas zone at each time point.

Returns The exhaust gas zone result object.


Returns the Thermo-Calc Console syntax formula used for calculating the activity of a component in the slag (e.g. AC(AL2O3, IONIC_LIQ) in a zone at each time point. The actual activity can be obtained using get_activity_of_slag().

Parameters

- zone – The zone object or the zone name
- component – The component
- reference – The reference for the activity, can be liquid or solid slag, default: liquid slag

Returns The formula for calculating the activity at each time point

get_formula_for_slag_property(zone: Union[tc_python.process_metallurgy.process.Zone, str], slag_property: tc_python.process_metallurgy.base.SlagProperty, slag_type: tc_python.process_metallurgy.base.SlagType = <SlagType.ALL: 2>) → List[str]

Returns the Thermo-Calc Console syntax formula used for calculating a property of the slag (e.g. B(CAO)/B(SIO2) in a zone at each time point. The actual slag property can be obtained using get_slag_property().

Parameters

- zone – The zone object or the zone name
- slag_property – The slag property
- slag_type – The part of the slag for which the property will be calculated. Can be all slag, the liquid or the solid slag. Default: all slag

Returns The formula for calculating the slag property at each time point

get_fraction_of_phase_groups(zone: Union[tc_python.process_metallurgy.process.Zone, str], unit: tc_python.utils.PhaseUnit = <PhaseUnit.MASS_FRACTION: 1>) → Dict[tc_python.process_metallurgy.base.PhaseGroup, List[float]]

Returns the fractions of the phase groups (e.g., all liquid slag) in a zone at each time point.

Parameters

- zone – The zone object or the zone name
- unit – The unit of the fraction
Returns The phase fractions at each time point

```python
def get_fraction_of_phases(zone: Union[tc_python.process_metallurgy.process.Zone, str], unit: tc_python.utils.PhaseUnit = <PhaseUnit.MASS_FRACTION: 1>) → Dict[str, List[float]]
```

Returns the fractions of all stable phases in a zone at each time point.

**Parameters**

- `zone` – The zone object or the zone name
- `unit` – The unit of the fraction

**Returns** The phase fractions at each time point

```python
def get_gas_components() → Set[str]
```

Returns all components of the gas phase defined for the elements present in the simulation.

**Returns** The components of the gas phase

```python
def get_num_of_performed_steps() → List[int]
```

Returns the accumulated number of performed time steps at each time point.

**Note:** The number of performed time steps can differ from the index of the time step in the result list because time steps might have been repeated with smaller step size during a process simulation.

**Returns** The accumulated number of performed time steps

```python
def get_oxygen_partial_pressure(zone: Union[tc_python.process_metallurgy.process.Zone, str]) → List[float]
```

Returns the partial pressure of oxygen in the zone at each time point.

**Parameters** `zone` – The zone object or the zone name

**Returns** The partial pressure [Pa]

```python
def get_pressure(zone: Union[tc_python.process_metallurgy.process.Zone, str]) → List[float]
```

Returns the pressure in a zone at each time point.

**Parameters** `zone` – The zone object or the zone name

**Returns** The pressure at each time point [Pa]

```python
def get_slag_property(zone: Union[tc_python.process_metallurgy.process.Zone, str], slag_property: tc_python.process_metallurgy.base.SlagProperty, slag_type: tc_python.process_metallurgy.base.SlagType = <SlagType.ALL: 2>) → List[float]
```

Returns a property of the slag in a zone at each time point. These properties are mostly used to describe the property of a slag to pick up sulfur.

**Parameters**

- `zone` – The zone object or the zone name
- `slag_property` – The slag property
- `slag_type` – The part of the slag for which the property will be calculated. Can be all slag, the liquid or the solid slag. **Default:** all slag

**Returns** The slag property at each time point [unit depending on the property]

```python
def get_stable_phases(zone: Union[tc_python.process_metallurgy.process.Zone, str]) → Set[str]
```

Returns the stable phases in a zone.
Parameters **zone** – The zone object or the zone name

Returns The stable phases

```python
get_stable_phases_in_phase_group(zone: Union[tc_python.process_metallurgy.process.Zone, str], phase_group: tc_python.process_metallurgy.base.PhaseGroup) → Set[str]
```

Returns the stable phases of a phase group (e.g., all solid slag) in a zone.

Parameters

• **zone** – The zone object or the zone name

• **phase_group** – The phase group

Returns The stable phases of the phase group

```python
get_temperature(zone: Union[tc_python.process_metallurgy.process.Zone, str]) → List[float]
```

Returns the temperature of a zone at each time point.

Parameters **zone** – The zone object or the zone name

Returns The temperature at each time point [K]

```python
get_time_points() → List[float]
```

Returns the time points of the process simulation. All result quantities are returned for exactly these time points.

Returns The time points [s]

```python
get_value_of(zone: Union[tc_python.process_metallurgy.process.Zone, str], classic_expression: str) → List[float]
```

Returns a value for a thermodynamic quantity in a zone at each time point.

**Warning:** It should normally not be required to use this method, use the appropriate method available in the API instead.

Parameters

• **zone** – The zone object or the zone name

• **classic_expression** – The thermodynamic quantity to get the value of in ThermoCalc Console Mode syntax (for example “NPM(FCC_A1)"

Returns The requested value at each time point

```python
get_viscosity_dynamic_of_phase(zone: Union[tc_python.process_metallurgy.process.Zone, str], phase: str) → List[float]
```

Returns the dynamic viscosity of a phase in a zone at each time point.

Parameters

• **zone** – The zone object or the zone name

• **phase** – The phase name

Returns The dynamic viscosity at each time point [Pa*s]

```python
get_viscosity_kinematic_of_phase(zone: Union[tc_python.process_metallurgy.process.Zone, str], phase: str) → List[float]
```

Returns the kinematic viscosity of a phase in a zone at each time point.

Parameters
• **zone** – The zone object or the zone name
• **phase** – The phase name

**Returns** The kinematic viscosity at each time point \([m^{2}/s]\)

```python
```

**Bases:** `tc_python.process_metallurgy.process.Zone`

A reaction zone in a process simulation, this is representing the interface layer between two bulk zones that are in contact and can react with each other, for example the steel melt and the top slag. The size of the reaction zone is dynamic and determined by the mass transfer coefficient. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

**add_addition** *(addition: tc_python.process_metallurgy.process.AbstractSingleTimeAddition, time: float = 0.0)*

Adds a single-time addition at the specified time point to the zone. The addition will be dissolved immediately.

**Parameters**

• **addition** – A `SingleTimeAddition` or `SingleTimeGasAddition`
• **time** – The time point [s]

**Returns** This `ReactionZone` object

**add_continuous_addition** *(addition: tc_python.process_metallurgy.process.AbstractContinuousAddition, from_time: float = 0.0, to_time: float = nan)*

Adds a constant addition continuously during the specified time period to the zone. All added material will be dissolved immediately.

**Parameters**

• **addition** – A `ContinuousAddition` or `ContinuousGasAddition`
• **from_time** – The start time point [s]
• **to_time** – The end time point [s]

**Returns** This `ReactionZone` object

**add_heat_transfer** *(heat_transfer_coefficient: float)*

Adds heat transfer through the reaction zone, i.e., between the two attached bulk zones.

**Parameters**

• **heat_transfer_coefficient** – The heat transfer coefficient \([W/(K*m^{2})]\)

**Returns** This `ReactionZone` object

**add_power** *(power: float, from_time: float = 0.0, to_time: float = nan)*

Adds a constant power during a specified time period to the zone (for example heating or cooling).

**Parameters**

• **power** – The power [MW]
• **from_time** – The start time point [s]
• **to_time** – The end time point [s]

**Returns** This `ReactionZone` object

*add_transfer_of_phase_group*(transfer_of_phase_group: tc_python.process_metallurgy.process.TransferOfPhaseGroup)

Adds transfer of a certain phase group through the reaction zone during each time step, i.e. from one of the attached bulk zones to the other. This is for example used to model inclusion flotation from the steel melt to the slag.

**Parameters** transfer_of_phase_group – The transfer of phase group configuration, can be time-dependent.

**Returns** This `ReactionZone` object

*disable_degassing*(

Disables degassing for this zone, i.e. all gas formed at any time step will be staying in this zone.

**Returns** This `ReactionZone` object

*enable_degassing*(

Enables degassing for this zone, i.e. any gas formed at any time step will be removed after that time step. This gas will be transferred into the exhaust gas zone. **This is the default.**

**Returns** This `ReactionZone` object

*get_elements*(

Returns the elements present in the zone. The elements are determined by the additions.

**Returns** The elements

*get_id*(

Returns the unique id of the zone. :return: The zone id

*is_degassing_enabled*(

Returns if degassing is enabled

**Returns** If degassing is enabled

```python
class tc_python.process_metallurgy.process.SingleTimeAddition (composition: Dict[str, float], amount: float, temperature: float = 293.15, composition_unit: tc_python.utils.CompositionUnit = <CompositionUnit.MASS_PERCENT: 0>, do_scale: bool = False)
```

**Bases:** `tc_python.process_metallurgy.process.AbstractSingleTimeAddition`

An addition in a process simulation that is added at a distinct time point.

It is assumed that the addition is dissolved instantaneously.

**Tip:** By setting `do_scale=True`, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not summing to 100% / 1. An example could be a slag addition which is provided like this: 90 wt-% CaO - 5 wt-% Al2O3 - 4 wt-% SiO2.
Parameters

- `composition` – The composition
- `amount` – The amount [kg]
- `temperature` – The initial addition temperature (default: 20 °C) [K]
- `composition_unit` – The composition unit
- `do_scale` – If the composition is scaled to 100% / fraction of 1

`get_amount()` → float

Returns the amount of this addition.

Returns The amount [kg]

`get_composition_unit()` → `tc_python.utils.CompositionUnit`

Returns the composition unit used in this addition.

Returns The composition unit

class `tc_python.process_metallurgy.process.SingleTimeGasAddition`

```python
class tc_python.process_metallurgy.process.SingleTimeGasAddition:
    def __init__(self, composition: Dict[str, float], amount: float, temperature: float = 293.15, amount_unit: tc_python.utils.GasAmountUnit = <GasAmountUnit.NORM_CUBIC_METER: 0>, composition_unit: tc_python.utils.GasCompositionUnit = <GasCompositionUnit.VOLUME_PERCENT: 4>, do_scale: bool = False):
```

Bases: `tc_python.process_metallurgy.process.AbstractSingleTimeAddition`

A gas addition in a process simulation that is added at a distinct time point.

It is assumed that the addition is dissolved instantaneously.

Tip: By setting `do_scale=True`, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not scaling to 100% / 1. An example could be a gas addition which is provided like this: 90 vol-% Ar - 10 vol-% O2.

Parameters

- `composition` – The composition
• **amount** – The amount
• **temperature** – The initial addition temperature (default: 20 °C) [K]
• **amount_unit** – The amount unit
• **composition_unit** – The composition unit
• **do_scale** – If the composition is scaled to 100% / fraction of 1

get_amount () → float
Returns the amount of this addition.

**Note:** The amount unit can be obtained using `get_amount_unit()`.

get_amount_unit () → tc_python.utils.GasAmountUnit
Returns the amount unit used in this addition.

Returns The amount unit

get_composition_unit () → tc_python.utils.GasCompositionUnit
Returns the composition unit used in this addition.

Returns The composition unit

```python
class tc_python.process_metallurgy.process.SlagBulkZone (density: float)
Bases: tc_python.process_metallurgy.process.Zone
A slag bulk zone in a process simulation.

This is representing a large volume in the process, for example the top slag. A zone is a volume in a process that
has identical temperature and composition. It has well-defined boundaries to other zones.

The name of this zone is automatically defined and unique.

add_addition (addition: tc_python.process_metallurgy.process.AbstractSingleTimeAddition, time: float = 0.0)
Adds a single-time addition at the specified time point to the zone. The addition will be dissolved imme-
diately.

Parameters
  • **addition** – A `SingleTimeAddition` or `SingleTimeGasAddition`
  • **time** – The time point [s]

Returns This `SlagBulkZone` object

add_continuous_addition (addition: tc_python.process_metallurgy.process.AbstractContinuousAddition, from_time: float = 0.0, to_time: float = nan)
Adds a constant addition continuously during the specified time period to the zone. All added material will
be dissolved immediately.

Parameters
  • **addition** – A `ContinuousAddition` or `ContinuousGasAddition`
  • **from_time** – The start time point [s]
  • **to_time** – The end time point [s]

Returns This `SlagBulkZone` object
```
add_power (power: float, from_time: float = 0.0, to_time: float = nan)

Adds a constant power during a specified time period to the zone (for example heating or cooling).

Parameters

- **power** – The power [MW]
- **from_time** – The start time point [s]
- **to_time** – The end time point [s]

Returns

This SlagBulkZone object

disable_degassing()

Disables degassing for this zone, i.e. all gas formed at any time step will be staying in this zone.

Returns

This SlagBulkZone object

enable_degassing()

Enables degassing for this zone, i.e. any gas formed at any time step will be removed after that time step. This gas will be transferred into the exhaust gas zone. **This is the default.**

Returns

This SlagBulkZone object

get_density () → float

Returns the density of the zone

Returns

The density [kg/m**3]

get_elements () → Set[str]

Returns the elements present in the zone. The elements are determined by the additions.

Returns

The elements

get_id () → str

Returns the unique id of the zone. :return: The zone id

get_phase_group_to_transfer () → tc_python.process_metallurgy.base.PhaseGroup

Returns the phase group that is transferred from the attached reaction zones back to this zone after each time step.

Returns

The phase group

is_degassing_enabled () → bool

Returns if degassing is enabled in the zone.

Returns

If degassing is enabled

class tc_python.process_metallurgy.process.TransferOfPhaseGroup (phase_group_to_transfer: tc_python.process_metallurgy.base.PhaseGroup, source_zone: tc_python.process_metallurgy.process.Zone)

Bases: object

The transfer of a percentage of a certain phase group (e.g., solid slag) between zones during each time step. This is for example used to model inclusion flotation from the steel melt to the slag.

add (transfer_rate: float, time: float = 0)

Adds the transfer rate valid beginning at a time point.

This value is valid until another value is defined for a later time point.

Parameters

- **transfer_rate** – The transfer rate [% of phase group amount/s]
- **time** – The time point where the transfer of a phase group begins to be valid [s]
Returns This TransferOfPhaseGroup object

get_phase_group_to_transfer() → tc_python.process_metallurgy.base.PhaseGroup
Returns the phase group to be transferred

get_transfer_source_zone_id() → str
The id of the source zone of the transfer

Returns This source zone id

class tc_python.process_metallurgy.process.Zone
Bases: object
The base class of a zone in a process simulation. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

add_addition(addition: tc_python.process_metallurgy.process.AbstractSingleTimeAddition, time: float = 0.0)
Adds a single-time addition at the specified time point to the zone. The addition will be dissolved immediately.

Parameters
• addition – A SingleTimeAddition or SingleTimeGasAddition
• time – The time point [s]

Returns This Zone object

add_continuous_addition(addition: tc_python.process_metallurgy.process.AbstractContinuousAddition, from_time: float = 0.0, to_time: float = nan)
Adds a constant addition continuously during the specified time period to the zone. All added material will be dissolved immediately.

Parameters
• addition – A ContinuousAddition or ContinuousGasAddition
• from_time – The start time point [s]
• to_time – The end time point [s]

Returns This Zone object

add_power(power: float, from_time: float = 0.0, to_time: float = nan)
Adds a constant power during a specified time period to the zone (for example heating or cooling).

Parameters
• power – The power [MW]
• from_time – The start time point [s]
• to_time – The end time point [s]

Returns This Zone object

disable_degassing()
Disables degassing for this zone, i.e. all gas formed at any time step will be staying in this zone.

Returns This Zone object

enable_degassing()
Enables degassing for this zone, i.e. any gas formed at any time step will be removed after that time step. This gas will be transferred into the exhaust gas zone. This is the default.

5.1. Calculations
Returns This Zone object

get_elements() \rightarrow \text{Set}[\text{str}]
Returns the elements present in the zone. The elements are determined by the additions.

Returns The elements

get_id() \rightarrow \text{str}
Returns the unique id of the zone. :return: The zone id

is_degassing_enabled() \rightarrow \text{bool}
Returns if degassing is enabled in the zone.

Returns If degassing is enabled

5.2 Module “system”

class tc_python.system.MultiDatabaseSystemBuilder(multi_database_system_builder)
Bases: object

Used to select databases, elements, phases etc. and create a System object. The difference to the class SystemBuilder is that the operations are performed on all the previously selected databases. The system is then used to create calculations.

create_and_select_species(stoichiometry: \text{str})
Specify a species from the already entered elements. The stoichiometry of the species is the chemical formula of the species. The created species will also be automatically selected.

Note: The elements in the chemical formula are normally separated by stoichiometric numbers. Neither parenthesis “()” nor an underscore “_” is allowed in the chemical formula, while the special combination “/-” or “/+” can be used. Consult the Thermo-Calc database documentation for details about the syntax.

Parameters  \textbf{stoichiometry} – The stoichiometry of the species

Returns This MultiDatabaseSystemBuilder object

deselect_constituent_on_sublattice(phase_name: \text{str}, sublattice_no: \text{int}, constituent_name_to_deselect: \text{str})
Rejects a constituent on a sublattice in a phase in both the thermodynamic and the kinetic database.

Parameters
•  \textbf{phase_name} – The name of the phase
•  \textbf{sublattice_no} – The number of the sublattice (starting with 1)
•  \textbf{constituent_name_to_deselect} – The name of the constituent to deselect

Returns This MultiDatabaseSystemBuilder object

deselect_phase(phase_name_to_deselect: \text{str})
Rejects a phase for both the thermodynamic and the kinetic database.

Parameters  \textbf{phase_name_to_deselect} – The phase name

Returns This MultiDatabaseSystemBuilder object

deselect_species(species_name: \text{str})
Removes the species from the system.
Parameters `species_name` – The species

Returns This `MultiDatabaseSystemBuilder` object

`get_system()` → `tc_python.system.System`

Creates a new System object that is the basis for all calculation types. Several calculation types can be defined later from the object; these are independent.

Returns A new `System` object

`select_constituent_on_sublattice(phase_name: str, sublattice_no: int, constituent_name_to_select: str)`

Selects a constituent on a sublattice in a phase in both the thermodynamic and the kinetic database.

Note: Previously the third parameter `constituent_name_to_select` had a wrong name, it has been corrected in version 2021b.

Parameters

- `phase_name` – The name of the phase
- `sublattice_no` – The number of the sublattice (starting with 1)
- `constituent_name_to_select` – The name of the constituent to select

Returns This `MultiDatabaseSystemBuilder` object

`select_phase(phase_name_to_select: str)`

Selects a phase for both the thermodynamic and the kinetic database.

Parameters `phase_name_to_select` – The phase name

Returns This `MultiDatabaseSystemBuilder` object

`select_species(species_name: str)`

Adds the species to the system. Up to 1000 species can be defined in a single system.

Parameters `species_name` – The species

Returns This `MultiDatabaseSystemBuilder` object

`with_new_composition_set(composition_set: tc_python.entities.CompositionSet)`

Used to enter two or more composition sets for a phase. If a phase has a miscibility gap it is necessary to have two composition sets, one for each possible composition that can be stable simultaneously.

The databases often create the typical composition sets for phases automatically when data are retrieved. The equilibrium calculations (using the default settings with global minimization) will usually add new composition sets if needed.

Note: Precipitation and diffusion calculations can require the user to define additional composition sets. E.g. in the case where the new composition set is needed in the configuration of the calculation.

Parameters `composition_set` – the composition set

Returns This `MultiDatabaseSystemBuilder` object

`without_default_phases()`

Rejects all the default phases from both the thermodynamic and the kinetic database, any phase now needs to be selected manually for the databases.

5.2. Module “system”
Returns This `MultiDatabaseSystemBuilder` object

class tc_python.system.System(system_instance)

    Bases: object

    A system containing selections for databases, elements, phases etc.

    Note: For the defined system, different calculations can be configured and run. **Instances of this class should always be created from a SystemBuilder.**

    Note: The system object is immutable, i.e. it cannot be changed after it has been created. If you want to change the system, you must instead create a new one.

convert_composition(input_composition: Dict[str, float], input_unit: tc_python.utils.ConversionUnit, output_unit: tc_python.utils.ConversionUnit, dependent_component: str = '') → Dict[str, float]

Provides conversion between composition units for any combination of chemical compounds. It is fast because no thermodynamic equilibrium calculation is involved.

**Syntax of the chemical compounds:** “Al2O3”, “FeO”, “CO”, “Fe”, “C”, …

**Note:** It is not required that the chemical compounds are components of the database. The only requirement is that all elements are present in the database.

Parameters

- **input_composition** – Composition (for example: \{“Al2O3”: 25.0, “FeO”: 75.0\})
- **input_unit** – Unit of the input composition
- **output_unit** – Requested output unit
- **dependent_component** – The dependent component (optional), for example: “Fe”.

    If no dependent component is specified the sum of the input composition needs to match 100% / 1

Returns The composition in the requested output unit

get_all_elements_in_databases() → List[str]

Returns the names of all elements present in the selected databases, regardless of the actual selection of elements.

Returns A list of element names

get_all_phases_in_databases() → List[str]

Returns all phase names present in the selected databases, regardless of selected elements, phases etc.

Returns A list of phase names

get_all_species_in_databases() → List[str]

Returns all species names present in the selected databases, regardless of the actual selection of elements, phases, …

Returns A list of species names
**get_element_object** *(element_name: str) → tc_python.entities.Element*

Returns the Element object of an element. This can be used to obtain detailed information about the element.

**Parameters**

*element_name* – The element name

**Returns**

A `Element` object

**get_elements_in_system** *(→ List[str]*)

Returns the names of all elements present in the selected system.

**Note:** The list does not contain any elements or components that have been auto-selected by the database(s) in a calculator. Use the `get_components()` of the calculator object instead to get the complete information.

**Returns**

A list of element names

**get_phase_object** *(phase_name: str) → tc_python.entities.Phase*

Returns the Phase object of a phase. This can be used to obtain detailed information about the phase.

**Parameters**

*phase_name* – The phase name

**Returns**

A `Phase` object

**get_phases_in_system** *(→ List[str]*)

Returns all phase names present in the system due to its configuration (selected elements, phases, etc.).

**Returns**

A list of phase names

**get_references** *(→ Dict[str, List[str]]*)

Provides a dictionary with database references per database in the selected system.

**Returns**

The database references

**get_species_in_system** *(→ List[str]*)

Returns the names of all species present in the selected system.

**Note:** The list does not contain any species or components that have been auto-selected by the database(s) in a calculator. Use the `get_components()` of the calculator object instead to get the complete information.

**Returns**

The list of species names

**get_species_object** *(species_name: str) → tc_python.entities.Species*

Returns the Species object of an species. This can be used to obtain detailed information about the species.

**Parameters**

*species_name* – The species name

**Returns**

A `Species` object

**get_system_data** *(→ tc_python.abstract_base.SystemData*)

Returns the content of the database. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

**Note:** Parameters can only be read from unencrypted (i.e. `user`) databases loaded as *.tdb*-file.
**Returns** The system data

```python
with_batch_equilibrium_calculation(default_conditions: bool = True,
components: List[str] = []) → tc_python.batch_equilibrium.BatchEquilibriumCalculation
```

Creates a batch-equilibrium calculation (a vectorized equilibrium calculation).

**Note:** Use this instead of looping if you want to calculate equilibria for a larger number of compositions and know the conditions in advance. This calculation type has improved performance when calculating a large number of equilibria when each individual calculation is quick. E.g. when evaluating single phase properties for thousands of compositions.

**Parameters**

- `default_conditions` – If `True`, automatically sets the conditions $N=1$ and $P=100000$
- `components` – Specify here the components of the system (for example: `[AL2O3, ...]`), *only necessary if they differ from the elements*. If this option is used, all elements of the system need to be replaced by a component.

**Returns** A new BatchEquilibriumCalculation object

```python
with_cct_precipitation_calculation() → tc_python.precipitation.PrecipitationCCTCalculation
```

Creates a CCT diagram calculation.

**Returns** A new PrecipitationCCTCalculation object

```python
with_isothermal_diffusion_calculation() → tc_python.diffusion.DiffusionIsoThermalCalculation
```

Creates an isothermal diffusion calculation.

**Returns** A new DiffusionIsoThermalCalculation object

```python
with_isothermal_precipitation_calculation() → tc_python.precipitation.PrecipitationIsoThermalCalculation
```

Creates an isothermal precipitation calculation.

**Returns** A new PrecipitationIsoThermalCalculation object

```python
with_material_to_material() → tc_python.material_to_material.MaterialToMaterialCalculationContainer
```

Provides access to all Material to Material calculations. The actual calculation needs to be chosen in the returned object.

**Returns** A new MaterialToMaterialCalculationContainer object

```python
with_non_isothermal_diffusion_calculation() → tc_python.diffusion.DiffusionNonIsoThermalCalculation
```

Creates a non-isothermal diffusion calculation.

**Returns** A new DiffusionNonIsoThermalCalculation object

```python
with_non_isothermal_precipitation_calculation() → tc_python.precipitation.PrecipitationNonIsoThermalCalculation
```

Creates a non-isothermal precipitation calculation.

**Returns** A new PrecipitationNonIsoThermalCalculation object

```python
with_phase_diagram_calculation(default_conditions: bool = True,
components: List[str] = []) → tc_python.step_or_map_diagrams.PhaseDiagramCalculation
```

Creates a phase diagram (map) calculation.

**Parameters**
• **default_conditions** – If True, automatically sets the conditions N=1 and P=100000

• **components** – Specify here the components of the system (for example: \([AL2O3, \ldots]\), *only necessary if they differ from the elements*. If this option is used, all elements of the system need to be replaced by a component.

**Returns** A new `PhaseDiagramCalculation` object

```python
with_property_diagram_calculation (default_conditions: bool = True,
                                   components: List[str] = []) ->
tc_python.step_or_map_diagrams.PropertyDiagramCalculation
```

Creates a property diagram (step) calculation.

**Parameters**

• **default_conditions** – If True, automatically sets the conditions N=1 and P=100000

• **components** – Specify here the components of the system (for example: \([AL2O3, \ldots]\), *only necessary if they differ from the elements*. If this option is used, all elements of the system need to be replaced by a component.

**Returns** A new `PropertyDiagramCalculation` object

```python
with_property_model_calculation (model: str,  
                                 path_to_models: str = '',  
                                 debug_model: bool = False) ->
tc_python.propertymodel.PropertyModelCalculation
```

Creates a Property Model calculation.

The parameter `debug_model` is only used when debugging self-developed models.

**Parameters**

• **model** – The Property Model to be calculated.

• **path_to_models** – The path where the Property Models are installed. If no value is entered, the Property Models folder used by the normal Thermo-Calc application is used.

• **debug_model** – Used when debugging self-developed models.

**Returns** A new `PropertyModelCalculation` object

```python
with_scheil_calculation () -> tc_python.scheil.ScheilCalculation
```

Creates a Scheil solidification calculation.

**Warning:** Scheil calculations do not support the GAS phase being selected, this means the `GAS` phase must always be deselected in the system if it is present in the database

**Returns** A new `ScheilCalculation` object

```python
with_single_equilibrium_calculation (default_conditions: bool = True,
                                     components: List[str] = []) ->
tc_python.single_equilibrium.SingleEquilibriumCalculation
```

Creates a single equilibrium calculation.

**Parameters**

• **default_conditions** – If True, automatically sets the conditions N=1 and P=100000
components – Specify here the components of the system (for example: \([\text{AL}2\text{O}3, \ldots]\)), only necessary if they differ from the elements. If this option is used, all elements of the system need to be replaced by a component.

Returns A new SingleEquilibriumCalculation object

with_ttt_precipitation_calculation() \(\rightarrow\) tc_python.precipitation.PrecipitationTTTCalculation
Creates a TTT diagram calculation.

Returns A new PrecipitationTTTCalculation object

class tc_python.system.SystemBuilder(system_builder)
Bases: object

Used to select databases, elements, phases etc. and create a System object. The system is then used to create calculations.

create_and_select_species(stoichiometry: str)
Specify a species from the already entered elements. The stoichiometry of the species is the chemical formula of the species. The created species will also be automatically selected.

Parameters

- **stoichiometry** – The stoichiometry of the species

Returns This SystemBuilder object

deselect_constituent_on_sublattice(phase_name: str, sublattice_no: int, constituent_name_to_deselect: str)
Rejects a constituent on a sublattice in a phase in the last specified database only.

Parameters

- **phase_name** – The name of the phase
- **sublattice_no** – The number of the sublattice (starting with 1)
- **constituent_name_to_deselect** – The name of the constituent to deselect

Returns This SystemBuilder object

deselect_phase(phase_name_to_deselect: str)
Rejects a phase in the last specified database only.

Parameters

- **phase_name_to_deselect** – The name of the phase

Returns This SystemBuilder object

deselect_species(stoichiometry: str)
Removes the species from the system.

Parameters

- **stoichiometry** – The species

Returns This SystemBuilder object

get_system() \(\rightarrow\) tc_python.system.System
Creates a new System object that is the basis for all calculation types. Several calculation types can be defined later from the object; these are independent.

Returns A new System object
get_system_for_scheil_calculations() → tc_python.system.System

Creates a new System object without gas phases being selected, that is the basis for all calculation types, but its particularly useful for Scheil solidification calculations, where the model does not allow that a gas phase is selected in the system.

Several calculation types can be defined later from the object; these are independent.

Note: Deprecated in version 2022b: Use get_system() instead. There are no longer any special requirements from the Thermo-Calc program to deselect the gas phase for Scheil. It will be removed in release 2023b.

Returns A new System object

select_constituent_on_sublattice(phase_name: str, sublattice_no: int, constituent_name_to_select: str)

Selects a constituent on a sublattice in a phase in the last specified database only.

Note: Previously the third parameter constituent_name_to_select had a wrong name, it has been corrected in version 2021b.

Parameters

• phase_name – The name of the phase
• sublattice_no – The number of the sublattice (starting with 1)
• constituent_name_to_select – The name of the constituent to select

Returns This SystemBuilder object

select_database_and_elements(database_name: str, list_of_element_strings: List[str])

Selects a thermodynamic or kinetic database and its selected elements (that will be appended). After that, phases can be selected or unselected.

Parameters

• database_name – The database name, for example “FEDEMO”
• list_of_element_strings – A list of one or more elements as strings, for example [“Fe”, “C”]

Returns This SystemBuilder object

select_phase(phase_name_to_select: str)

Selects a phase in the last specified database only.

Parameters phase_name_to_select – The name of the phase

Returns This SystemBuilder object

select_species(stoichiometry: str)

Adds the species to the system. Up to 1000 species can be defined in a single system.

Parameters stoichiometry – The species

Returns This SystemBuilder object
select_user_database_and_elements(path_to_user_database: str, list_of_element_strings: List[str])

Selects a thermodynamic database which is a user-defined database and select its elements (that will be appended).

**Note:** By using a r-literal, it is possible to use slashes on all platforms, also on Windows: select_user_database_and_elements(r"my path/user_db.tdb", ["Fe", "Cr"])

Otherwise it is required to use double backslashes on Windows as separator.

**Note:** On Linux and Mac the path is case-sensitive, also the file ending.

**Parameters**

- **path_to_user_database** – The path to the database file ("database".TDB), defaults to the current working directory. Only the filename is required if the database is located in the same folder as the script.

- **list_of_element_strings** – A list of one or more elements as strings, for example ["Fe", "C"]

**Returns** This SystemBuilder object

with_new_composition_set(composition_set: tc_python.entities.CompositionSet)

Used to enter composition sets for a phase. If a phase has a miscibility gap it is necessary to have two composition sets, one for each possible composition that can be stable simultaneously.

**Parameters** composition_set – The composition set

**Returns** This SystemBuilder object

without_default_phases()

Rejects all default phases in the last specified database only, any phase needs now to be selected manually for that database.

**Returns** This SystemBuilder object

### 5.3 Module “entities”

**class** tc_python.entities.CompositionSet(phase_name: str)

**Bases:** object

Used by the method tc_python.system.SystemBuilder.with_new_composition_set() to enter two or more composition sets for a phase.

**Parameters** phase_name – The name of the phase for which a new composition set is required

**set_major_constituents_for_sublattice(sublattice_index: int, major_constituents: List[str])**

Specify the new major constituent(s) for the sublattice.

**Default:** If not specified, a default is automatically chosen based on the specified composition set.

**Note:** This is useful in order to make calculations converge faster and more easily (because it may simplify giving start values when calculating the equilibrium as those phases with miscibility gaps should
have different major constituents for each composition set). The databases often set major constituents for several phases automatically when the data is retrieved.

## Parameters

- **sublattice_index** – Index of the sublattice to set the major constituents for (starting with 1)
- **major_constituents** – Optional list of the major constituents, which must be selected from the phase constitution of the current system.

## Returns

This `CompositionSet` object

### class tc_python.entities.Element(element)

Bases: object

Represents an element, making detailed information about the element accessible.

- **get_enthalpy**() → float
  Returns the enthalpy of the element at 298 K, part of the stable element reference state (SER).
  
  **Returns** The enthalpy [J]

- **get_entropy_diff_0_to_298k**() → float
  Returns the entropy difference 0 - 298 K of the element, part of the stable element reference state (SER).
  
  **Returns** The entropy difference 0 - 298 K [J/K]

- **get_molar_mass**() → float
  Returns the molar mass of the element.
  
  **Returns** The molar mass [g/mol]

- **get_name**() → str
  Returns the name of the element.
  
  **Returns** The element name

- **get_stable_element_reference**() → str
  Returns the stable element reference (i.e. the stable phase at 298.15 K and 1 bar, reference for all element thermodynamic data).
  
  **Returns** The name of the stable element reference

- **is_interstitial**() → bool
  Returns if the element is interstitial.
  
  **Returns** If the element is interstitial

- **is_special**() → bool
  Returns if the element is special (i.e. vacancies (VA) and electrons (denoted either as /- in gaseous, liquid or solid phases, or ZE in an aqueous solution phase)).
  
  **Returns** If the element is special

- **is_valid**() → bool
  Returns if the element is valid. Non-valid elements are represented by an empty name.
**Returns** If the element is valid

class tc_python.entities.Phase(phase)
   Bases: object

   Represents a phase, making detailed information about the phase accessible.

   get_name() → str
      Returns the name of the phase.

   Returns The phase name

   get_species() → Set[tc_python.entities.Species]
      Returns the species of the phase.

   Returns A set containing the species

   get_species_for_composition_profile() → Set[tc_python.entities.Species]
      Returns all species that need to be defined in a composition profile of the phase for diffusion simulations - except for one species that needs to be the dependent species.

   Note: In a composition profile of a phase for diffusion simulations it is necessary to specify all non-stoichiometric and non-special species. In case of a DILUTE diffusion model, the database enforces the choice of a certain dependent species.

   Returns Set with the species

   get_sublattices() → List[tc_python.entities.Sublattice]
      Returns the sublattices of the phase in a well-defined contiguous order.

   Returns A list containing the Sublattice objects

   get_type() → tc_python.entities.PhaseType
      Returns the type of the phase (liquid, ionic liquid, solid, gas).

   Returns The type of a phase

   has_diffusion_data() → bool
      Returns if diffusion data exists for the phase.

   Returns If diffusion data exists for the phase

   has_molar_volume_data() → bool
      Returns if molar volume data exists for the phase.

   Returns If molar volume data exists for the phase

   is_dilute_diffusion_model() → bool
      Returns if diffusion is described using the DILUTE model for the phase. This will always return False if no diffusion data is available.

   Returns If the DILUTE model is used

   is_gas() → bool
      Returns if the phase is a gas phase.

   Returns If the phase is a gas phase

   is_ionic_liquid() → bool
      Returns if the phase is an ionic liquid phase.

   Returns If the phase is an ionic liquid phase
is_liquid() → bool
    Returns if the phase is a liquid or ionic liquid phase.

    Returns  If the phase is a liquid phase

is_solid() → bool
    Returns if the phase is a solid phase.

    Returns  If the phase is a solid phase

class tc_python.entities.PhaseType(value)
    Bases: enum.Enum
    The type of a phase.

    GAS = 0
    Gas phase.

    IONIC_LIQUID = 2
    Ionic liquid phase.

    LIQUID = 1
    Liquid phase.

    SOLID = 3
    Solid phase.

class tc_python.entities.Species(species)
    Bases: object
    Represents a species, making detailed information about the species accessible.

    get_all_elements() → List[Tuple[tc_python.entities.Element, float]]
    Returns all the elements that the species is composed of.

    Returns  List of all elements of the species and their stoichiometry

    get_charge() → int
    Returns the charge of the species.

    Returns  The charge of the species

    get_name() → str
    Returns the name of the species.

    Returns  The species name

    is_element() → bool
    Returns if the species actually represents an element.

    Returns  If the species represents an element

    isInterstitial() → bool
    Returns if the species is interstitial.

    Returns  If the species is interstitial

Note: In the diffusion simulations (DICTRA), the assumption that the volume is carried by the substitu-
tional elements only is applied. The interstitial elements are assumed to have zero molar volumes.
is_special() → bool
Returns if the species is special (i.e. vacancies (VA) and electrons (denoted either as - in gaseous, liquid or solid phases, or ZE in an aqueous solution phase)).

Returns If the species is special

is_valid() → bool
Returns if the species is valid. Non-valid species are represented by an empty name.

Returns If the species is valid

to_element() → tc_python.entities.Element
Returns the Element representation of the species - if the species actually represents an element.

Returns The Element object

class tc_python.entities.Sublattice(sublattice)
Bases: object
Represents a sublattice of a phase.

get_constituents() → Set[tc_python.entities.Species]
Returns the constituents of the sublattice.

Returns A set containing the constituents

get_nr_of_sites() → float
Returns the number of sites in the sublattice.

Returns A float number

5.4 Module “server”

class tc_python.server.LoggingPolicy(value)
Bases: enum.Enum
Logging policy that determines how the TC-Python logs are presented to the user.

FILE = 1
Logging to a file.

NONE = 2
No logging at all.

SCREEN = 0
Logging to the screen.

class tc_python.server.MetallurgyCalculations(metallurgy_calculations)
Bases: object
Provides access to the calculation objects for all Process Metallurgy calculations.

These are specialised calculations for working with metallurgical processes. Both equilibrium calculations and kinetic process simulations (Effective Equilibrium Reaction Zone model) are available.

Creates an adiabatic equilibrium calculation for Process Metallurgy.

Parameters database – The thermodynamic database used in the calculation

Returns A new AdiabaticEquilibriumCalculation object
with_adiabatic_process_calculation(database: tc_python.process_metallurgy.base.ProcessDatabase)
          → tc_python.process_metallurgy.process.ProcessSimulationCalculation
Creates an adiabatic kinetic process simulation (EERZ, i.e. Effective Equilibrium Reaction Zone model).

Parameters database – The thermodynamic database used in the calculation

Returns A new ProcessSimulationCalculation object

with_isothermal_equilibrium_calculation(database: tc_python.process_metallurgy.base.ProcessDatabase)
          → tc_python.process_metallurgy.equilibrium.IsoThermalEquilibrium
Creates an isothermal equilibrium calculation for Process Metallurgy.

Parameters database – The thermodynamic database used in the calculation

Returns A new IsoThermalEquilibriumCalculation object

class tc_python.server.ResultLoader(result_loader)
Bases: object
Contains methods for loading results from previously done calculations.

diffusion(path: str) → tc_python.diffusion.DiffusionCalculationResult
Loads a DiffusionCalculationResult from disc.

Parameters path – path to the folder where result was previously saved.

Returns A new DiffusionCalculationResult object which later can be used to get specific values from the calculated result

phase_diagram(path: str) → tc_python.step_or_map_diagrams.PhaseDiagramResult
Loads a PhaseDiagramResult from disc.

Parameters path – path to the folder where result was previously saved.

Returns A new PhaseDiagramResult object which later can be used to get specific values from the calculated result

precipitation_TTT_or_CCT(path: str) → tc_python.precipitation.PrecipitationCalculationTTTorCCTResult
Loads a PrecipitationCalculationTTTorCCTResult from disc.

Parameters path – path to the folder where result was previously saved.

Returns A new PrecipitationCalculationTTTorCCTResult object which later can be used to get specific values from the calculated result

precipitation_single(path: str) → tc_python.precipitation.PrecipitationCalculationSingleResult
Loads a PrecipitationCalculationSingleResult from disc.

Parameters path – path to the folder where result was previously saved.

Returns A new PrecipitationCalculationSingleResult object which later can be used to get specific values from the calculated result

property_diagram(path: str) → tc_python.step_or_map_diagrams.PropertyDiagramResult
Loads a PropertyDiagramResult from disc.

Parameters path – path to the folder where result was previously saved.

Returns A new PropertyDiagramResult object which later can be used to get specific values from the calculated result

property_model(path: str) → tc_python.propertymodel.PropertyModelResult
Loads a PropertyModelResult from disc.

Parameters path – path to the folder where result was previously saved.
Returns  A new PropertyModelResult object which later can be used to get specific values from the calculated result

`scheil` *(path: str) → tc_python.scheil.ScheilCalculationResult*

Loads a ScheilCalculationResult from disc.

**Parameters**  path – path to the folder where result was previously saved.

**Returns**  A new ScheilCalculationResult object which later can be used to get specific values from the calculated result

`single_equilibrium` *(path: str) → tc_python.single_equilibrium.SingleEquilibriumResult*

Loads a SingleEquilibriumResult from disc.

**Parameters**  path – path to the folder where result was previously saved.

**Returns**  A new SingleEquilibriumResult object which later can be used to get specific values from the calculated result

---

**class**  tc_python.server.SetUp *(debug_logging=False)*

**Bases:**  object

Starting point for all calculations.

**Note:**  This class exposes methods that have no precondition, it is used for choosing databases and elements.

`disable_caching()`

A previously set cache folder is no longer used.

**Note:**  Within the session, caching is activated and used through the default temporary directory.

**Returns**  This SetUp object

`get_database_info` *(database_short_name: str) → str*

Obtains the short information available for the specified database.

**Parameters**  database_short_name – The name of the database (i.e. “FEDEMO”, . . . )

**Returns**  The short information about the database

`get_database_path_on_disk` *(database_short_name: str) → str*

Obtains the path to the database file on disk. `TCPATH` is a placeholder for the root path of the used Thermo-Calc installation.

**Note:**  Encrypted databases (*.TDC) cannot be edited.

**Parameters**  database_short_name – The name of the database (i.e. “FEDEMO”, . . . )

**Returns**  The path to the database on disk

`get_databases()` → List[str]

Obtains the short names of all databases available in the used Thermo-Calc installation.

**Note:**  Only databases with a valid license are listed.
**Returns** List of the available databases

`get_property_models(path_to_models: str = '') → Set[str]`

Lists the names of all Property Models in the specified directory.

If the directory is not specified, the Property Model folder used by the normal Thermo-Calc application is used.

**Parameters**
- `path_to_models` – The path where the Property Models are installed. If no value is entered, the Property Model folder used by the normal Thermo-Calc application is used.

**Returns** Set containing all Property Model names

`load_result_from_disk() → tc_python.server.ResultLoader`

Loads a previously calculated result from disk.

**Note:** This only works for results created by calling one of the save_result() methods on a Result class created from a calculation.

**Returns** A new ResultLoader object

`select_database_and_elements(database_name: str, list_of_elements: List[str]) → tc_python.system.SystemBuilder`

Selects a first thermodynamic or kinetic database and selects the elements in it.

**Parameters**
- `database_name` – The name of the database, for example “FEDEMO”
- `list_of_elements` – The list of the selected elements in that database, for example [“Fe”, “C”]

**Returns** A new SystemBuilder object

`select_thermodynamic_and_kinetic_databases_with_elements(thermodynamic_db_name: str, kinetic_db_name: str, list_of_elements: List[str]) → tc_python.system.MultiDatabaseSystemBuilder`

Selects the thermodynamic and kinetic database at once, guarantees that the databases are added in the correct order. Further rejection or selection of phases applies to both databases.

**Parameters**
- `thermodynamic_db_name` – The thermodynamic database name, for example “FEDEMO”
- `kinetic_db_name` – The kinetic database name, for example “MFEDEMO”
- `list_of_elements` – The list of the selected elements in that database, for example [“Fe”, “C”]

**Returns** A new MultiDatabaseSystemBuilder object

`select_user_database_and_elements(path_to_user_database: str, list_of_elements: List[str]) → tc_python.system.SystemBuilder`

Selects a user-defined database and selects the elements in it.
Note: By using a r-literal, it is possible to use slashes on all platforms, also on Windows: `select_user_database_and_elements(r"my path/user_db.tdb", ["Fe", "Cr"])

Otherwise it is required to use double backslashes on Windows as separator.

Note: On Linux and Mac the path is case-sensitive, also the file ending.

Parameters

- **path_to_user_database** – The path to the database file (“database”.TDB), defaults to the current working directory. Only filename is required if the database is located in the same folder as the script.

- **list_of_elements** – The list of the selected elements in that database, for example ["Fe", "C"]

Returns A new SystemBuilder object

```python
set_cache_folder(path: str = '', precision_for_floats: int = 12)
```

Sets a folder where results from calculations and state of systems are saved. If at any time a calculation is run which has the exact same setting as a previous, the calculation is not re-run. The result is instead loaded from this folder.

Note: The same folder can be used in several scripts, and it can even be shared between different users. It can be a network folder.

Parameters

- **path** – path to the folder where results should be stored. It can be relative or absolute.

- **precision_for_floats** – The number of significant figures used when comparing if the calculation has the same setting as a previous.

Returns This SetUp object

```python
set_ges_version(version: int = 6)
```

Setting the version of the Gibbs Energy System (GES).

Parameters **version** – The GES-version (currently version 5 or 6)

Returns This SetUp object

```python
set_log_level_to_debug()
```

Sets log level to DEBUG

Returns This SetUp object

```python
set_log_level_to_info()
```

Sets log level to INFO

Returns This SetUp object

```python
with_metallurgy() → tc_python.server.MetallurgyCalculations
```

Provides access to the calculation objects for all Process Metallurgy calculations.
These are specialised calculations for working with metallurgical processes. Both equilibrium calculations and kinetic process simulations (Effective Equilibrium Reaction Zone model) are available.

```python
class tc_python.server.TCPython(
    logging_policy=<LoggingPolicy.SCREEN: 0>,
    log_file=None, debug_mode=False, debug_logging=False,
    do_throw_on_backend_hard_crash=True, port_number=0
)
Bases: object
```

Starting point of the API. Typical syntax:

```python
with TCPython() as session:
    session.select_database_and_elements(...)  
```

**Note:** Each usage of `with TCPython()` causes significant overhead (starting a new process, stopping the old one, cleaning up the temporary disk space). Usually it is recommendable to call `with TCPython()` only once for each process, even if working in a loop. Instead you should pass the session or calculator object into the loop and use them there.

If necessary, beginning from version 2019a it is however possible to call `with TCPython` safely multiple times.

```python
tc_python.server.start_api_server(
    logging_policy=<LoggingPolicy.SCREEN: 0>,
    log_file=None, debug_mode=False, is_unittest=False,
    do_throw_on_backend_hard_crash=True, port_number=0
)
```

Starts a process of the API server and sets up the socket communication with it.

**Parameters**

- **logging_policy** – Determines if the TC-Python log output is sent to the screen (`LoggingPolicy.SCREEN`), to file (`LoggingPolicy.FILE`) or nothing is logged at all (`LoggingPolicy.NONE`). **Default:** `LoggingPolicy.SCREEN`. Note that the log-handlers can also be adapted through the `tc_python.LOGGER` object at any time.

- **log_file** – The log-file relative to the current path or absolute, only relevant if `logging_policy=LoggingPolicy.FILE`. Log-output will be appended.

- **debug_mode** – If `True` it is tried to open a connection to an already running API-server. This is only used for debugging the API itself.

- **is_unittest** – Should be `True` if called by a unit test, **only to be used internally for development**.

- **do_throw_on_backend_hard_crash** – If `True` an `UnrecoverableCalculationException` will be thrown if the Java-backend crashes hard, if `False` the application will simply crash with a FORTRAN-stacktrace. If `True` the exception can be caught outside of the `with`-clause and the application can continue, if `False` more information about the error is shown by the stacktrace.

- **port_number** – The port number for the communication with the Java-backend server. **This is not required to be changed by normal users.**

**Warning:** Most users should use `TCPython` using a with-statement for automatic management of the resources (network sockets and temporary files). If you anyway need to use that method, make sure to call `stop_api_server()` in any case using the try-finally-pattern.
tc_python.server.start_matlab_server(logging_policy=<LoggingPolicy.SCREEN: 0>, log_file=None, debug_mode=False, is_unittest=False, do_throw_on_backend_hard_crash=True, port_number=0)

tc_python.server.stop_api_server(gateway_id: str = None)

Clears all resources used by the session (i.e. shuts down the API server and deletes all temporary files). The disk usage of temporary files might be significant.

**Warning:** Call this method only if you used `start_api_server()` initially. It should never be called when the API has been initialized in a with-statement using TCFPython.

### 5.5 Module “quantity_factory”

**class** tc_python.quantity_factory.DiffusionQuantity

**Bases:** tc_python.quantity.AbstractQuantity

Factory class providing quantities used for defining diffusion simulations and their results.

**Note:** In this factory class only the most common quantities are defined, you can always use the Console Mode syntax strings in the respective methods as an alternative (for example: “NPM(*)”).

**classmethod** activity_of_component(component: *str*, use_ser: *bool* = False) → tc_python.quantity.ActivityOfComponent

Creates a quantity representing the activity of a component.

**Parameters**

- **component** – The name of the component, use `ALL_COMPONENTS` to choose all components
- **use_ser** – Use Stable-Element-Reference(SER). The user-defined reference state is be used if this setting is set to `False`.

**Returns** A new `ActivityOfComponent` object.


Creates a quantity representing the chemical diffusion coefficient of a phase [m²/s].

**Parameters**

- **phase** – The name of the phase
- **diffusing_element** – The diffusing element
- **gradient_element** – The gradient element
- **reference_element** – The reference element (for example “Fe” in a steel)

**Returns** A new `ChemicalDiffusionCoefficient` object.

**classmethod** chemical_potential_of_component(component: *str*, use_ser: *bool* = False) → tc_python.quantity.ChemicalPotentialOfComponent

Creates a quantity representing the chemical potential of a component [J].
Parameters

• **component** – The name of the component, use `ALL_COMPONENTS` to choose all components

• **use_ser** – Use Stable-Element-Reference(SER). The user-defined reference state is used if this setting is set to `False`.

**Returns** A new `ChemicalPotentialOfComponent` object.

classmethod distance(region: str = 'All') -> tc_python.quantity.Distance

Creates a quantity representing the distance [m].

**Parameters**

• **region** – The name of the region or `All` to choose global.


Creates a quantity representing the intrinsic diffusion coefficient of a phase [m^2/s].

**Parameters**

• **phase** – The name of the phase

• **diffusing_element** – The diffusing element

• **gradient_element** – The gradient element

• **reference_element** – The reference element (for example “Fe” in a steel)

**Returns** A new `IntrinsicDiffusionCoefficient` object.


Creates a quantity representing L^2 of a phase [m^2/s].

**Parameters**

• **phase** – The name of the phase

• **diffusing_element** – The diffusing element

• **gradient_element** – The gradient element

• **reference_element** – The reference element (for example “Fe” in a steel)

**Returns** A new `Lbis` object.

classmethod mass_fraction_of_a_component(component: str) -> tc_python.quantity.MassFractionOfAComponent

Creates a quantity representing the mass fraction of a component.

**Parameters**

• **component** – The name of the component or `ALL_COMPONENTS` to choose all components

**Returns** A new `MassFractionOfAComponent` object.

classmethod mass_fraction_of_a_phase(phase: str) -> tc_python.quantity.MassFractionOfAPhase

Creates a quantity representing the mass fraction of a phase.

**Parameters**

• **phase** – The name of the phase or `ALL_PHASES` to choose all phases.

**Returns** A new `MassFractionOfAPhase` object.
class method mobility_of_component_in_phase(\texttt{phase: str, component: str}) \rightarrow \texttt{tc\_python.quantity.MobilityOfComponentInPhase}

Creates a quantity representing the mobility of a component in a phase [m^2/Js].

**Parameters**

- \texttt{phase} – The name of the phase
- \texttt{component} – The name of the component

**Returns** A new \texttt{MobilityOfComponentInPhase} object.

class method mole_fraction_of_a_component(\texttt{component: str}) \rightarrow \texttt{tc\_python.quantity.MoleFractionOfAComponent}

Creates a quantity representing the mole fraction of a component.

**Parameters** \texttt{component} – The name of the component or \texttt{ALL\_COMPONENTS} to choose all components

**Returns** A new \texttt{MoleFractionOfAComponent} object.

class method mole_fraction_of_a_phase(\texttt{phase: str}) \rightarrow \texttt{tc\_python.quantity.MoleFractionOfAPhase}

Creates a quantity representing the mole fraction of a phase.

**Parameters** \texttt{phase} – The name of the phase or \texttt{ALL\_PHASES} to choose all phases

**Returns** A new \texttt{MoleFractionOfAPhase} object.

class method position_of_lower_boundary_of_region(\texttt{region: str}) \rightarrow \texttt{tc\_python.quantity.PositionOfLowerBoundaryOfRegion}

Creates a quantity representing the position of lower boundary of a region [m].

**Parameters** \texttt{region} – The name of the region

**Returns** A new \texttt{PositionOfLowerBoundaryOfRegion} object.

class method position_of_upper_boundary_of_region(\texttt{region: str}) \rightarrow \texttt{tc\_python.quantity.PositionOfUpperBoundaryOfRegion}

Creates a quantity representing the position of upper boundary of a region [m].

**Parameters** \texttt{region} – The name of the region

**Returns** A new \texttt{PositionOfUpperBoundaryOfRegion} object.

class method temperature() \rightarrow \texttt{tc\_python.quantity.Temperature}

Creates a quantity representing the temperature [K].

**Returns** A new \texttt{Temperature} object.


Creates a quantity representing thermodynamic factor of a phase.

**Parameters**

- \texttt{phase} – The name of the phase
- \texttt{diffusing_element} – The diffusing element
- \texttt{gradient_element} – The gradient element
- \texttt{reference_element} – The reference element (for example “Fe” in a steel)

**Returns** A new \texttt{ThermoDynamicFactor} object.
classmethod **time**() \(\rightarrow\) tc_python.quantity.Time

Creates a quantity representing the time [s].

classmethod **total_mass_fraction_of_component** (**component**: str) \(\rightarrow\) tc_python.quantity.TotalMassFractionOfComponent

Creates a quantity representing the total mass fraction of a component.

**Parameters**

- **component** – The name of the component

**Returns**

A new TotalMassFractionOfComponent object.

classmethod **total_mass_fraction_of_component_in_phase** (**phase**: str, **component**: str) \(\rightarrow\) tc_python.quantity.TotalMassFractionOfComponentInPhase

Creates a quantity representing the total mass fraction of a component in a phase.

**Parameters**

- **phase** – The name of the phase
- **component** – The name of the component

**Returns**

A new TotalMassFractionOfComponentInPhase object.

classmethod **total_mass_fraction_of_phase** (**phase**: str) \(\rightarrow\) tc_python.quantity.TotalMassFractionOfPhase

Creates a quantity representing the total mass fraction of a phase.

**Parameters**

- **phase** – The name of the phase

**Returns**

A new TotalMassFractionOfPhase object.

classmethod **total_mole_fraction_of_component** (**component**: str) \(\rightarrow\) tc_python.quantity.TotalMoleFractionOfComponent

Creates a quantity representing the total mole fraction of a component.

**Parameters**

- **component** – The name of the component

**Returns**

A new TotalMoleFractionOfComponent object.

classmethod **total_mole_fraction_of_component_in_phase** (**phase**: str, **component**: str) \(\rightarrow\) tc_python.quantity.TotalMoleFractionOfComponentInPhase

Creates a quantity representing the total mole fraction of a component in a phase.

**Parameters**

- **phase** – The name of the phase
- **component** – The name of the component

**Returns**

A new TotalMoleFractionOfComponentInPhase object.

classmethod **total_volume_fraction_of_phase** (**phase**: str) \(\rightarrow\) tc_python.quantity.TotalVolumeFractionOfPhase

Creates a quantity representing the total volume fraction of a phase.

**Parameters**

- **phase** – The name of the phase

**Returns**

A new TotalVolumeFractionOfPhase object.

classmethod **tracer_diffusion_coefficient** (**phase**: str, **diffusing_element**: str) \(\rightarrow\) tc_python.quantity.TracerDiffusionCoefficient

Creates a quantity representing tracer diffusion coefficient of a phase [m²/s].

**Parameters**

- **phase** – The name of the phase
• **diffusing_element** – The diffusing element

**Returns** A new TracerDiffusionCoefficient object.

```python
@classmethod
diffusing_element
```

Creates a quantity representing the diffusing element.

**Parameters**

- `component`: The name of the component

```python
component: str
```

**Returns** A new TracerDiffusionCoefficient object.

```python
→ tc_python.quantity.UFractionOfAComponent
```

**Parameters**

- `expression`: The function expression

```python
expression: str
```

**Returns** A new Function object.

```python
→ tc_python.quantity.VelocityOfLowerBoundaryOfRegion
```

**Parameters**

- `region`: The name of the region

```python
region: str
```

**Returns** A new VelocityOfLowerBoundaryOfRegion object.

```python
→ tc_python.quantity.VelocityOfUpperBoundaryOfRegion
```

**Parameters**

- `region`: The name of the region

```python
region: str
```

**Returns** A new VelocityOfUpperBoundaryOfRegion object.

```python
→ tc_python.quantity.WidthOfRegion
```

**Parameters**

- `region`: The name of the region

```python
region: str
```

**Returns** A new WidthOfRegion object.

```python
class tc_python.quantity_factory.IndependentVariable
```

Factory class providing quantities used for defining the independent variable in general diffusion result querying.

```python
classmethod distance
```

Creates an independent variable representing the distance [m].

**Returns** A new Distance object

```python
classmethod time
```

Creates an independent variable representing the time [s].

**Returns** A new Time object

```python
class tc_python.quantity_factory.PlotCondition
```

Factory class providing quantities used for defining the plot condition in general diffusion result querying.

**Note:** In this factory class only the most common quantities are defined, you can always use the Console Mode syntax strings in the respective methods as an alternative (for example: “time last”).
**classmethod distance** *(distancepoint: float, region: str = 'All') → tc_python.quantity.DistanceCondition*

Creates a plot condition representing the distance [m).

**Changes in version 2019b:** Mandatory parameter *distancepoint* added

**Parameters**

- **distancepoint** – The distance from the lower interface of the region
- **region** – The name of the region or *All* to choose global.

**Returns** A new *DistanceCondition* object

**classmethod integral** () → tc_python.quantity.IntegralCondition

Creates an integral plot condition.

**Returns** A new *IntegralCondition* object

**classmethod interface** *(region: str, interface_position: tc_python.utils.InterfacePosition) → tc_python.quantity.InterfaceCondition*

Creates a plot condition representing an interface between two regions.

**Parameters**

- **region** – The name of the region used for defining the interface
- **interface_position** – The position of the interface relative to that region (lower or upper)

**Returns** A new *InterfaceCondition* object

**classmethod time** *(timepoint: Union[float, str] = 'Last') → tc_python.quantity.TimeCondition*

Creates a plot condition representing the time [s].

**Change in version 2019b:** Lists of timepoints are no longer supported

**Parameters** timepoint – The timepoint. Optionally “Last” can be used for the end of the simulation

**Returns** A new *TimeCondition* object

**class tc_python.quantity_factory.ScheilQuantity**

Bases: tc_python.quantity.AbstractQuantity

Factory class providing quantities used for defining a Scheil calculation result *(tc_python.scheil.ScheilCalculationResult).*

**classmethod apparent_heat_capacity_per_gram** () → tc_python.quantity.ApparentHeatCapacityPerGram

Creates a quantity representing the apparent heat capacity [J/g/K].

**Returns** A new *ApparentHeatCapacityPerGram* object.

**classmethod apparent_heat_capacity_per_mole** () → tc_python.quantity.ApparentHeatCapacityPerMole

Creates a quantity representing the apparent heat capacity [J/mol/K].

**Returns** A new *ApparentHeatCapacityPerMole* object.

**classmethod apparent_volumetric_thermal_expansion_coefficient** () → tc_python.quantity.ApparentVolumetricThermalExpansionCoefficient

Creates a quantity representing the apparent volumetric thermal expansion coefficient of the system [1/K].

**Returns** A new *ApparentVolumetricThermalExpansionCoefficient* object.
classmethod composition_of_phase_as_mole_fraction(phase: str, component: str) → tc_python.quantity.CompositionOfPhaseAsMoleFraction

Creates a quantity representing the composition of a phase [mole-fraction].

Parameters

• phase – The name of the phase, use ALL_PHASES to choose all stable phases

• component – The name of the component, use ALL_COMPONENTS to choose all components

Returns A new CompositionOfPhaseAsMoleFraction object.

classmethod composition_of_phase_as_weight_fraction(phase: str, component: str) → tc_python.quantity.CompositionOfPhaseAsWeightFraction

Creates a quantity representing the composition of a phase [weight-fraction].

Parameters

• phase – The name of the phase, use ALL_PHASES to choose all stable phases

• component – The name of the component, use ALL_COMPONENTS to choose all components

Returns A new CompositionOfPhaseAsWeightFraction object.

classmethod density_of_phase(phase: str) → tc_python.quantity.DensityOfPhase

Creates a quantity representing the average density of a phase [g/cm^3].

Parameters phase – The name of the phase or ALL_PHASES to choose all phases

Returns A new DensityOfPhase object.

classmethod density_of_solid_phase(phase: str) → tc_python.quantity.DensityOfSolidPhase

Creates a quantity representing the average density of a solid phase [g/cm^3].

Note: Deprecated in version 2022a: This quantity has been renamed to density_of_phase(). It will be removed in release 2023a.

Parameters phase – The name of the phase or ALL_PHASES to choose all solid phases

Returns A new DensityOfSolidPhase object.

classmethod density_of_system() → tc_python.quantity.DensityOfSystem

Creates a quantity representing the average density of the system [g/cm^3].

Returns A new DensityOfSystem object.


Creates a quantity representing the (molar) fraction of the specified component being present in the specified phase compared to the overall system [-]. This corresponds to the degree of segregation to that phase.

Parameters

• phase – The name of the phase

• component – The name of the component

Returns A new DistributionOfComponentOfPhase object.
**classmethod heat_per_gram**() → tc_python.quantity.HeatPerGram

Creates a quantity representing the total heat release from the liquidus temperature down to the current temperature [J/g].

**Note:** The total or apparent heat release during the solidification process consists of two parts: one is the so-called latent heat, i.e. heat due to the liquid -> solid phase transformation (**latent_heat_per_mole()** and **latent_heat_per_gram()**), and the other is the heat related to the specific heat of liquid and solid phases (**heat_per_mole()** and **heat_per_gram()**).

**Returns** A new **HeatPerGram** object.

**classmethod heat_per_mole**() → tc_python.quantity.HeatPerMole

Creates a quantity representing the total heat release from the liquidus temperature down to the current temperature [J/mol].

**Note:** The total or apparent heat release during the solidification process consists of two parts: one is the so-called latent heat, i.e. heat due to the liquid -> solid phase transformation (**latent_heat_per_mole()** and **latent_heat_per_gram()**), and the other is the heat related to the specific heat of liquid and solid phases (**heat_per_mole()** and **heat_per_gram()**).

**Returns** A new **HeatPerMole** object.

**classmethod latent_heat_per_gram**() → tc_python.quantity.LatentHeatPerGram

Creates a quantity representing the cumulated latent heat release from the liquidus temperature down to the current temperature [J/g].

**Note:** The total or apparent heat release during the solidification process consists of two parts: one is the so-called latent heat, i.e. heat due to the liquid -> solid phase transformation (**latent_heat_per_mole()** and **latent_heat_per_gram()**), and the other is the heat related to the specific heat of liquid and solid phases (**heat_per_mole()** and **heat_per_gram()**).

**Returns** A new **LatentHeatPerGram** object.

**classmethod latent_heat_per_mole**() → tc_python.quantity.LatentHeatPerMole

Creates a quantity representing the cumulated latent heat release from the liquidus temperature down to the current temperature [J/mol].

**Note:** The total or apparent heat release during the solidification process consists of two parts: one is the so-called latent heat, i.e. heat due to the liquid -> solid phase transformation (**latent_heat_per_mole()** and **latent_heat_per_gram()**), and the other is the heat related to the specific heat of liquid and solid phases (**heat_per_mole()** and **heat_per_gram()**).

**Returns** A new **LatentHeatPerMole** object.

**classmethod mass_fraction_of_a_solid_phase**(*phase: str*) → tc_python.quantity.MassFractionOfASolidPhase

Creates a quantity representing the mass fraction of a solid phase.

**Parameters phase** – The name of the phase or **ALL_PHASES** to choose all solid phases.
classmethod mass_fraction_of_all_liquid() → tc_python.quantity.MassFractionOfAllLiquid
    Creates a quantity representing the total mass fraction of all the liquid phase.
    Returns A new MassFractionOfAllLiquid object.

classmethod mass_fraction_of_all_solid_phases() → tc_python.quantity.MassFractionOfAllSolidPhase
    Creates a quantity representing the total mass fraction of all solid phases.
    Returns A new MassFractionOfAllSolidPhase object.

classmethod molar_volume_of_phase(phase: str) → tc_python.quantity.MolarVolumeOfPhase
    Creates a quantity representing the molar volume of a phase [m^3/mol].
    Parameters phase – The name of the phase or ALL_PHASES to choose all phases
    Returns A new MolarVolumeOfPhase object.

classmethod molar_volume_of_system() → tc_python.quantity.MolarVolumeOfSystem
    Creates a quantity representing the molar volume of the system [m^3/mol].
    Returns A new MolarVolumeOfSystem object.

classmethod mole_fraction_of_a_solid_phase(phase: str) → tc_python.quantity.MoleFractionOfASolidPhase
    Creates a quantity representing the molar fraction of a solid phase.
    Parameters phase – The name of the phase or ALL_PHASES to choose all solid phases
    Returns A new MoleFractionOfASolidPhase object.

classmethod mole_fraction_of_all_liquid() → tc_python.quantity.MoleFractionOfAllLiquid
    Creates a quantity representing the total molar fraction of all the liquid phase.
    Returns A new MoleFractionOfAllLiquid object.

classmethod mole_fraction_of_all_solid_phases() → tc_python.quantity.MoleFractionOfAllSolidPhases
    Creates a quantity representing the total molar fraction of all solid phases.
    Returns A new MoleFractionOfAllSolidPhases object.

classmethod site_fraction_of_component_in_phase(phase: str, component: str, sub_lattice_ordinal_no: int = 0) → tc_python.quantity.SiteFractionOfComponentInPhase
    Creates a quantity representing the site fractions [-].
    Parameters
    • phase – The name of the phase, use ALL_PHASES to choose all stable phases
    • component – The name of the component, use ALL_COMPONENTS to choose all components
    • sub_lattice_ordinal_no – The ordinal number (i.e. 1, 2, …) of the sublattice of interest, use None to choose all sublattices

Note: Detailed information about the sublattices can be obtained by getting the Phase object of a phase from the System object using tc_python.system.System.get_phase_in_system. For each phase the sublattices are obtained by using tc_python.system.Phase.get_sublattices. The
order in the returned list is equivalent to the sublattice ordinal number expected, **but note that the ordinal numbers start with 1.**

Returns A new SiteFractionOfComponentInPhase object.

classmethod temperature() → tc_python.quantity.Temperature

  Creates a quantity representing the temperature [K].

  Returns A new Temperature object.

class tc_python.quantity_factory.ThermodynamicQuantity

Bases: tc_python.quantity.AbstractQuantity

Factory class providing quantities used for defining equilibrium calculations (single equilibrium, property and phase diagrams, ...) and their results.

**Note:** In this factory class only the most common quantities are defined, you can always use the *Console Mode* syntax strings in the respective methods as an alternative (for example: “NPM(*)”).

classmethod activity_of_component(component: str, use_ser: bool = False) → tc_python.quantity.ActivityOfComponent

  Creates a quantity representing the activity of a component [-].

  Parameters

  • **component** – The name of the component, use ALL_COMPONENTS to choose all components

  • **use_ser** – Use Stable-Element-Reference(SER). The user-defined reference state is used if this setting is set to False.

  Returns A new ActivityOfComponent object.


  Creates a quantity representing the chemical diffusion coefficient of a phase [m^2/s].

  Parameters

  • **phase** – The name of the phase

  • **diffusing_element** – The diffusing element

  • **gradient_element** – The gradient element

  • **reference_element** – The reference element (for example “Fe” in a steel)

  Returns A new ChemicalDiffusionCoefficient object.

classmethod chemical_potential_of_component(component: str, use_ser: bool = False) → tc_python.quantity.ChemicalPotentialOfComponent

  Creates a quantity representing the chemical potential of a component [J].

  Parameters

  • **component** – The name of the component, use ALL_COMPONENTS to choose all components
• **use_ser** – Use Stable-Element-Reference(SER). The user-defined reference state is used if this setting is set to `False`.

Returns A new `ChemicalPotentialOfComponent` object.

classmethod composition_of_phase_as_mole_fraction (phase: str, component: str = 'All') →
tc_python.quantity.CompositionOfPhaseAsMoleFraction

Creates a quantity representing the composition of a phase [mole-fraction].

Parameters

• **phase** – The name of the phase, use `ALL_PHASES` to choose all stable phases
• **component** – The name of the component, use `ALL_COMPONENTS` to choose all components

Returns A new `CompositionOfPhaseAsMoleFraction` object.

classmethod composition_of_phase_as_weight_fraction (phase: str, component: str) →
tc_python.quantity.CompositionOfPhaseAsWeightFraction

Creates a quantity representing the composition of a phase [weight-fraction].

Parameters

• **phase** – The name of the phase, use `ALL_PHASES` to choose all stable phases
• **component** – The name of the component, use `ALL_COMPONENTS` to choose all components

Returns A new `CompositionOfPhaseAsWeightFraction` object.

classmethod gibbs_energy_of_a_phase (phase: str, use_ser: bool = False) →
tc_python.quantity.GibbsEnergyOfAPhase

Creates a quantity representing the Gibbs energy of a phase [J].

Parameters

• **phase** – The name of the phase or `ALL_PHASES` to choose all phases
• **use_ser** – Use Stable-Element-Reference(SER). The user-defined reference state will be used when this setting is set to False.

Returns A new `GibbsEnergyOfAPhase` object.

classmethod mass_fraction_of_a_component (component: str) →
tc_python.quantity.MassFractionOfAComponent

Creates a quantity representing the mass fraction of a component.

Parameters **component** – The name of the component or `ALL_COMPONENTS` to choose all components

Returns A new `MassFractionOfAComponent` object.

classmethod mass_fraction_of_a_phase (phase: str) →
tc_python.quantity.MassFractionOfAPhase

Creates a quantity representing the mass fraction of a phase.

Parameters **phase** – The name of the phase or `ALL_PHASES` to choose all phases.

Returns A new `MassFractionOfAPhase` object.

classmethod mole_fraction_of_a_component (component: str) →
tc_python.quantity.MoleFractionOfAComponent

Creates a quantity representing the mole fraction of a component.
Parameters **component** – The name of the component or **ALL_COMPONENTS** to choose all components

Returns A new **MoleFractionOfAComponent** object.

**classmethod mole_fraction_of_a_phase** (**phase**: `str`) →

tc_python.quantity.MoleFractionOfAPhase

Creates a quantity representing the mole fraction of a phase.

Parameters **phase** – The name of the phase or **ALL_PHASES** to choose all phases

Returns A new **MoleFractionOfAPhase** object.

**classmethod normalized_driving_force_of_a_phase** (**phase**: `str`) →

tc_python.quantity.NormalizedDrivingForceOfAPhase

Creates a quantity representing normalized driving force of a phase [-].

**Warning:** A driving force calculation requires that the respective phase has been set to the state **DORMANT**. The parameter **All** is only reasonable if all phases have been set to that state.

Parameters **phase** – The name of the phase or **ALL_PHASES** to choose all phases

Returns A new **DrivingForceOfAPhase** object.

**classmethod pressure** () → tc_python.quantity.Pressure

Creates a quantity representing the pressure [Pa].

Returns A new **Pressure** object.

**classmethod system_size** () → tc_python.quantity.SystemSize

Creates a quantity representing the system size [mol].

Returns A new **SystemSize** object.

**classmethod temperature** () → tc_python.quantity.Temperature

Creates a quantity representing the temperature [K].

Returns A new **Temperature** object.

**classmethod tracer_diffusion_coefficient** (**phase**: `str`, **diffusing_element**: `str`) →

tc_python.quantity.TracerDiffusionCoefficient

Creates a quantity representing tracer diffusion coefficient of a phase [m^2/s].

Parameters

- **phase** – The name of the phase

- **diffusing_element** – The diffusing element

Returns A new **TracerDiffusionCoefficient** object.

**classmethod u_fraction_of_a_component** (**component**: `str`) →

tc_python.quantity.UFractionOfAComponent

Creates a quantity representing the u-fraction of a component.

Parameters **component** – The name of the component

Returns A new **UFractionOfAComponent** object.

**classmethod user_defined_function** (**expression**: `str`) →

tc_python.quantity.Function

Creates a quantity representing a user-defined function.

Parameters **expression** – The function expression
Returns a new `Function` object

```python
classmethod volume_fraction_of_a_phase(phase: str) →
```

Creates a quantity representing the volume fraction of a phase.

**Parameters**

- `phase` – The name of the phase or `ALL_PHASES` to choose all phases

**Returns**

A new `VolumeFractionOfAPhase` object.

## 5.6 Module “utils”

```python
class tc_python.utils.CompositionType(value)
```

Bases: `enum.Enum`

The type of composition.

**COMPONENT** = 1
Composition given per component, this will be identical to `ELEMENT` in case of metals.

**ELEMENT** = 0
Composition given per element.

```python
class tc_python.utils.CompositionUnit(value)
```

Bases: `enum.Enum`

The composition unit.

**MASS_FRACTION** = 1
Mass fraction.

**MASS_PERCENT** = 0
Mass percent.

**MOLE_FRACTION** = 3
Mole fraction.

**MOLE_PERCENT** = 2
Mole percent.

```python
class tc_python.utils.ConversionUnit(value)
```

Bases: `enum.Enum`

The composition unit used in a conversion.

**MOLE_FRACTION** = 0
Mole fraction.

**MOLE_PERCENT** = 1
Mole percent.

**WEIGHT_FRACTION** = 2
Weight fraction.

**WEIGHT_PERCENT** = 3
Weight percent.

```python
class tc_python.utils.GasAmountUnit(value)
```

Bases: `enum.Enum`

The amount of a gas.
KILOGRAM = 1
    Kilogram.

NORM_CUBIC_METER = 0
    Norm cubic meter (according to ISO 2533, p=101325 Pa, T=288.15 K, typically used for trading of gas bottles). Other definitions vary only slightly.

class tc_python.utils.GasCompositionUnit(value)
    Bases: enum.Enum

    The composition unit for a gas.

    MASS_FRACTION = 1
        Mass fraction.

    MASS_PERCENT = 0
        Mass percent.

    MOLE_FRACTION = 3
        Mole fraction.

    MOLE_PERCENT = 2
        Mole percent.

    VOLUME_FRACTION = 5
        Volume fraction.

    VOLUME_PERCENT = 4
        Volume percent.

class tc_python.utils.GasRateUnit(value)
    Bases: enum.Enum

    The rate of a gas flow.

    KILOGRAM_PER_SEC = 1
        Kilogram per second.

    NORM_CUBIC_METER_PER_SEC = 0
        Norm cubic meter per second (according to ISO 2533, p=101325 Pa, T=288.15 K, typically used for trading of gas bottles). Other definitions vary only slightly.

class tc_python.utils.InterfacePosition(value)
    Bases: enum.Enum

    The position of an interface relative to its region. Only used for diffusion simulations.

    LOWER = 0
        The interface is on the lower side of its region.

    UPPER = 1
        The interface is on the upper side of its region.

class tc_python.utils.PhaseUnit(value)
    Bases: enum.Enum

    The units available for a phase fraction.

    MASS_FRACTION = 1
        Mass fraction.

    MOLE_FRACTION = 0
        Mole fraction.
VOLUME_FRACTION = 2
Volume fraction.

class tc_python.utils.ResultValueGroup(result_line_group_java)
Bases: object
A x-y-dataset representing a line data calculation result (i.e. a Thermo-Calc \textit{quantity 1} vs. \textit{quantity 2}).

\textbf{Warning:} Depending on the calculator, the dataset might contain \textit{NaN}-values to separate the data between different subsets.

Variables

- \textbf{label} – a str describing what the data corresponds to
- \textbf{x} – list of floats representing the first quantity (“x-axis”)
- \textbf{y} – list of floats representing the second quantity (“y-axis”)

\texttt{get\_label} () \rightarrow \texttt{str}
Accessor for the line label: return the line label

\texttt{get\_x} () \rightarrow \texttt{List[\texttt{float}]}
Accessor for the x-values: return the x values

\texttt{get\_y} () \rightarrow \texttt{List[\texttt{float}]}
Accessor for the y-values: return the y values

class tc_python.utils.TemperatureProfile
Bases: object
Represents a time-temperature profile used by non-isothermal calculations.

\textbf{Note:} The total simulation time can differ from the defined temperature profile. Constant temperature is assumed for any timepoint after the end of the defined profile.

\texttt{add\_time\_temperature} (time: float, temperature: float)
Adds a time-temperature point to the non-isothermal temperature profile.

\textbf{Parameters}

- \textbf{time} – The time [s]
- \textbf{temperature} – The temperature [K]

\textbf{Returns} This \texttt{TemperatureProfile} object

5.7 Module “propertymodel_sdk”

class tc_python.propertymodel_sdk.CCTResult(quantity_id: str, description: str)
Bases: tc_python.propertymodel_sdk.ResultQuantity
Represents a Continuous Cooling (CCT) result.

\textbf{Parameters}

- \textbf{quantity_id} – The id of this result
• `description` – The description of this result

`add_time_temperature (time_temperature_id: str, description: str)`

Adds a time-temperature pair to the result.

Parameters

• `time_temperature_id` – The id of the time-temperature pair

• `description` – The description of the time-temperature pair

`temperature_suffix = ' (T)'`

The temperature suffix of a `CCTResult`

`time_suffix = ' (t)'`

The time suffix of a `CCTResult`

```python
class tc_python.propertymodel_sdk.CCTResultValues (cooling_rate: float = -1.0, cooling_rate_start_temperature: float = -1.0, cooling_rate_end_temperature: float = -1.0)
```

Bases: object

Represents Continuous Cooling (CCT) result values.

Parameters

• `cooling_rate` – The cooling rate [K/s]

• `cooling_rate_start_temperature` – The start temperature of cooling [K]

• `cooling_rate_end_temperature` – The end temperature of cooling [K]

`set_result_time_temperature (time_temperature_id: str, time: float, temperature: float)`

Sets a time-temperature pair of the result.

Parameters

• `time_temperature_id` – The id of the time-temperature pair

• `time` – The time [s]

• `temperature` – The temperature [K]

```python
class tc_python.propertymodel_sdk.CalculationContext (system: tc_python.system.System, model_utils=None)
```

Bases: object

Represents the interface of the Property Model with the Thermo-Calc application and the rest of the TC-Python functionality.

Parameters

• `system` – The system object of this calculation

• `model_utils` – The model utils object

`get_argument_ids ()` → `Set[str]`

Returns a list with the arguments of the models. Including arguments created from dynamic parameters.

`get_dependent_component ()` → `str`

Obtains the dependent component from the UI
**Note:** The dependent component is that which has no composition specified explicitly, typically this is the major element of the material (such as Fe, Al, Ni, ...)

**Returns**  The dependent component

```python
def get_mass_fractions() -> Dict[str, float]
```
Obtains the current composition from the UI as mass-fraction.

**Note:** In case of stepping over one or multiple axis, the returned data will represent the composition at the current step.

**Returns**  The composition (key: component, value: content) [mass-fraction]

```python
def get_mass_percents() -> Dict[str, float]
```
Obtains the current composition from the UI in mass-percent.

**Note:** In case of stepping over one or multiple axis, the returned data will represent the composition at the current step.

**Returns**  The composition (key: component, value: content) [mass-percent]

```python
def get_mole_fractions() -> Dict[str, float]
```
Obtains the current composition from the UI as mole-fraction.

**Note:** In case of stepping over one or multiple axis, the returned data will represent the composition at the current step.

**Returns**  The composition (key: component, value: content) [mole-fraction]

```python
def get_mole_percents() -> Dict[str, float]
```
Obtains the current composition from the UI in mole-percent.

**Note:** In case of stepping over one or multiple axis, the returned data will represent the composition at the current step.

**Returns**  The composition (key: component, value: content) [mole-percent]

```python
def get_temperature() -> float
```
Obtains the current temperature from the UI.

**Returns**  The temperature [K]

```python
def get_ui_boolean_value(component_id: str) -> bool
```
Obtains the value from the specified checkbox UI component.

**Parameters**  component_id – Id of the checkbox

**Returns**  The setting of the checkbox
get_ui_condition_list (component_id: str) → tc_python.propertymodel_sdk.ConditionListEntry
Used to get the selected condition from components of type UIConditionListComponent
:param component_id: Id of the list UI component
:return: The selected condition

get_ui_float_value (component_id: str) → float
Obtains the value from the specified UI component.

Parameters
component_id – Id of the UI component

Returns
The value

get_ui_list_value (component_id: str) → str
Obtains the selected entry from a UI component list. If a special element (such as ANY, NONE, ...) is
selected, the corresponding locale-independent placeholder is provided.

Parameters
component_id – Id of the list UI component

Returns
The selected entry

get_ui_string_value (component_id: str) → str
Obtains the selected entry from a UI component text field.

Parameters
component_id – Id of the string UI component

Returns
The selected entry

get_ui_temperature_value (component_id: str) → float
Obtains the temperature from the specified temperature UI component.

Parameters
component_id – Id of the temperature UI component

Returns
The temperature [K], note that input unit of the UI is specified in the model panel. If
required, the temperature is automatically converted to K.

set_result_cct_values (quantity_id: str, r: tc_python.propertymodel_sdk.CCTResultValues)
Sets the value of a previously defined result quantity (of type CCTResultValues) for further usage in
the Thermo-Calc application for plotting, etc.

Parameters
- quantity_id – unique id of the result quantity
- r – the CCTResultValues to be set

set_result_quantity_value (quantity_id: str, value: float, parameter: str = '')
Sets the value of a previously defined result quantity for further usage in the Thermo-Calc application for
plotting, etc.

Note: Any result quantity that remains unset is automatically set to NaN.

Parameters
- quantity_id – Unique id of the result quantity
- parameter – Use if result is parameterized. f.i. “per phase”
- value – The value to be set

set_single_equilibrium_result (quantity_id: str, r: tc_python.single_equilibrium.SingleEquilibriumResult)

class tc_python.propertymodel_sdk.ConditionListEntry
Bases: object
Used in combination with components of type UIConditionListComponent.

Contains the element, if the selected condition is a composition. Contents the Console Mode syntax of the selected condition. Contains the unit of the selected condition.

class tc_python.propertymodel_sdk.PropertyModel(_locale: str = 'en-US')
    Bases: object

    The abstract base class for all property models.

**Note:** Every Property Model needs to implement most of the abstract methods of this class. However, some abstract methods are optional and should only be implemented if required.

**Note:** If overwriting the constructor in a Property Model, the constructor of the implemented class must have the identical signature and should pass the parameters to this base class constructor.

**Tip:** It is possible to switch off *internal INFO-log messages coming from the calculation engine* by changing the log-level on the TC-Python log object like this: `logging.getLogger("tc_python").setLevel(logging.ERROR)`.

Parameters

- **_locale** – The locale to be used, this is an internal parameter and is of no meaning to the end-user

Variables

- **logger** – logger object that is connected to the Thermo-Calc UI (INFO- and WARNING-level will be printed as INFO, ERROR-level as ERROR), it can be accessed like this: `self.logger.info("Some message")`

**abstract** `add_button_callback` (component_id: str, ui_component_ids: Dict[str, int]) → List[tc_python.propertymodel_sdk.UIComponent]

Implement this method if you have one or more UI components on which you called `UIComponent.enable_add_button()`, which adds a + button next to the component.

This method will be executed when you press any such + button.

This method is typically used to add more UI components dynamically and the method must return a list of the UI components to be added.

This method can optionally be implemented by a Property Model.

Parameters

- **component_id** – The id of the UI component next to the pressed + button
- **ui_component_ids** – dict with the id:s and the index of the current ui components of the model. Including arguments created in previous calls to `add_button_callback`.

Returns A list of `UIComponent` objects to be added

**abstract** `after_evaluations`()

Called by the Thermo-Calc application immediately after the last model evaluation (using the method `PropertyModel.evaluate_model()`). Use this method for any required cleanup.

This method can optionally be implemented by a Property Model.

**abstract** `before_evaluations` (context: tc_python.propertymodel_sdk.CalculationContext)

Called by the Thermo-Calc application immediately before the first model evaluation (using the method `PropertyModel.evaluate_model()`). Use this method for any required preparations.
This method can optionally be implemented by a Property Model.

Parameters context – The calculation context

abstract evaluate_model (context: tc_python.propertymodel_sdk.CalculationContext)
Called by the Thermo-Calc application when the model should be actually calculated. This is the main-method of the Property Model that contains the actual calculation code.

This method needs to be implemented by all property models.

Parameters context – The calculation context, this provides access to the Thermo-Calc application and all other TC-Python modules

abstract get_license_key () -> str
Provides the license key of the model.

This method can optionally be implemented by a Property Model.

abstract provide_calculation_result_quantities () -> List[tc_python.propertymodel_sdk.ResultQuantity]
Called by the Thermo-Calc application when the model should provide its result quantity objects.

This method needs to be implemented by all property models.

Returns Result quantity objects of the model (to be filled later with results in the method PropertyModel.evaluate_model())

abstract provide_model_category () -> List[str]
Called by the Thermo-Calc application when the model should provide its category (shown in the Thermo-Calc model tree).

This method needs to be implemented by all property models.

Returns Category of the model, it may be present in several categories

abstract provide_model_description () -> str
Called by the Thermo-Calc application when the model should provide its detailed description.

This method needs to be implemented by all property models.

Returns Description text for the model

abstract provide_model_name () -> str
Called by the Thermo-Calc application when the model should provide its name (shown in the Thermo-Calc model tree).

This method needs to be implemented by all property models.

Returns Name of the model

abstract provide_model_parameters () -> Dict[str, float]
Called by the Thermo-Calc application when the model should provide all model parameters and their current values.

This method can optionally be implemented by a Property Model.

Note: These are internal variables of the Property Model that are intended to be modified from the outside. Typically this is used to adjust their values in a optimizer during the development of the model.

Returns The model parameter ids and their current values [unit according to the parameter meaning]
abstract **provide_ui_panel_components** () → List[tc_python.propertymodel_sdk.UIComponent]
Called by the Thermo-Calc application when the model should provide its UI components for the model panel to be plotted. This happens also whenever a model gets checked in the model tree.

**This method needs to be implemented by all property models.**

**Returns** Model UI panel components in the order to be presented in the model panel

abstract **remove_button_callback** (component_id: str, ui_component_ids: Dict[str, int]) → List[str]
Implement this method if you have one or more UI components on which you called UIComponent.enable_remove_button(), which adds a - button next to the component.

This method will be executed when you press any such - button.

This method is typically used to remove UI components dynamically and the method must return a list of the ids of the components that are going to be removed.

**This method can optionally be implemented by a Property Model.**

**Parameters**

- **component_id** – the id of the UI component next to the pressed - button
- **ui_component_ids** – list with the id:s and the index of the current ui components of the model. Including arguments created in previous calls to add_button_callback.

**Returns** a list of UI component ids that are required to be removed

abstract **set_model_parameter** (model_parameter_id: str, value: float)
Called by the Thermo-Calc application when a model parameter should be reset.

**This method can optionally be implemented by a Property Model.**

**Note:** These are internal variables of the Property Model that are intended to be modified from the outside. Typically this is used to adjust their values in a optimizer during the development of the model.

**Parameters**

- **model_parameter_id** – The parameter id
- **value** – The value [unit according to the parameter meaning]

class tc_python.propertymodel_sdk.ResultQuantity (quantity_id: str, description: str, quantity_type: tc_python.propertymodel_sdk.ResultQuantityType)

**Bases:** object

Defines a calculation result quantity of a Property Model that is identified by a unique id.

**Parameters**

- **quantity_id** – Unique id of the quantity
- **description** – Description of the quantity (shown in the Thermo-Calc UI)
- **quantity_type** – Type of the quantity (defines the unit)

**get_description** () → str
Obtains the description of the quantity.

**Returns** Description of the quantity
get_id() → str
Obtains the id of the quantity.

Returns Unique id of the quantity

get_type() → tc_python.propertymodel_sdk.ResultQuantityType
Obtains the type of quantity.

Returns Type of the quantity

class tc_python.propertymodel_sdk.ResultQuantityType(value)
Bases: enum.Enum
Defining the type of a result quantity.

CCT_QUANTITY = 5
A cct quantity

ENERGY_QUANTITY = 2
An energy quantity

GENERAL_QUANTITY = 0
A general quantity

LENGTH_QUANTITY = 7
A length in quantity

SINGLE_EQUILIBRIUM_QUANTITY = 6
A cct quantity

STRENGTH_QUANTITY = 8
A strength quantity

SURFACE_ENERGY_QUANTITY = 3
A surface energy quantity

TEMPERATURE_QUANTITY = 1
A temperature quantity

TIME_QUANTITY = 4
A time quantity

class tc_python.propertymodel_sdk.SpecialListMarkers
Bases: object
Placeholders for special list elements that are locale-dependent. They will be provided by UI list components if a special marker has been selected.

ANY_LIST_MARKER = 'ANY'
Marker that represents “Any”

NONE_LIST_MARKER = 'NONE'
Marker that represents “None”

class tc_python.propertymodel_sdk.UIBooleanComponent(component_id: str, name: str, description: str, setting: bool)
Bases: tc_python.propertymodel_sdk.UIComponent
Checkbox UI component of the model panel.

Parameters

• component_id – Unique id of the component

• name – Name of the component, will be presented in the model panel
connect_component_visibility(dependent_component_id: str)
Connects the visibility of any other UI component of the model panel to the value of this boolean component.

Parameters
dependent_component_id – Id of the UI element to be dependent on this boolean component

enable_add_button()
Adds a + button to the right of the UI component.

Returns This UI component

enable_remove_button()
Adds a - button to the right of the UI component.

Returns This UI component

get_dependent_components() → List[str]
Obtains a list containing all UI elements currently connected regarding their visibility.

Returns A list with the component id of all UI elements currently connected

get_setting() → bool
Obtains the setting of the checkbox.

Returns The setting of the checkbox

remove_component_visibility(dependent_component_id: str)
Removes the visibility connection to a UI component that has been previously connected.

Parameters
dependent_component_id – Id of the previously connection UI element

get_index(index: int = -1)
Sets the position in the graphical user interface.

Parameters index – The position

Returns This UI component

class tc_python.propertymodel_sdk.UIComponent(component_id: str, name: str, description: str)
Bases: object

Abstract Base class for all UI components of the model panel.

Never make an instance of UIComponent, always use the sub-classes. For instance UIStringComponent.

Parameters

• component_id – Unique id of the component

• name – Name of the component, will be presented in the model panel

• description – Additional description of the component

get_description() → str
Obtains the additional description of the component.

Returns Additional description of the component

get_id() → str
Obtains the unique id of the component.

Returns Unique id of the component
get_name() → str
Obtains the name of the component.

Returns Name of the component, will be presented in the model panel

class tc_python.propertymodel_sdk.UIConditionListComponent(component_id: str, name: str, description: str)
Bases: tc_python.propertymodel_sdk.UIComponent
System condition list UI component of the model panel.

Parameters
- component_id – Unique id of the component
- name – Name of the component, will be presented in the model panel
- description – Additional description of the component

class tc_python.propertymodel_sdk.UIFloatComponent(component_id: str, name: str, description: str, value: float)
Bases: tc_python.propertymodel_sdk.UIComponent
General real value text field UI component of the model panel.

Parameters
- component_id – Unique id of the component
- name – Name of the component, will be presented in the model panel
- description – Additional description of the component
- value – Initial setting of the text field

enable_add_button()  
Adds a + button to the right of the UI component.

Returns This UI component

enable_remove_button()  
Adds a - button to the right of the UI component.

Returns this UI component

get_value() → float  
Obtains the setting of the text field.

Returns The setting of the text field

set_index(index: int = -1)
Sets the position in the graphical user interface.

Parameters index – The position

Returns This UI component

class tc_python.propertymodel_sdk.UIGeneralListComponent(component_id: str, name: str, description: str, content: List[Tuple[str, str]], selected_entry: str = '')
Bases: tc_python.propertymodel_sdk.UIComponent
General list UI component of the model panel that can contain any strings.

Parameters
• **component_id** – Unique id of the component
• **name** – Name of the component, will be presented in the model panel
• **description** – Additional description of the component
• **content** – Entries of the list, they need to contain a locale-independent id and a localized content string, for example: ["ENTRY_1_ID", "entry 1"], [ENTRY_2_ID", "entry 2"]
• **selected_entry** – Entry to be initially selected. If omitted, by default the first element is selected.

**connect_component_visibility** *(dependent_component_id: str, selected_item_to_set_visible: str)*

Connects the visibility of any other UI component of the model panel to the selection of a certain entry of the list.

**Parameters**

- **dependent_component_id** – Id of the UI element to be dependent on the chosen element
- **selected_item_to_set_visible** – Entry (locale independent id) of the list to be chosen to set the dependent component visible

**enable_add_button** ()

Adds a + button to the right of the UI component.

**Returns** This UI component

**enable_remove_button** ()

Adds a - button to the right of the UI component.

**Returns** This UI component

**get_content** () → List[Tuple[str, str]]

Obtains the entries of the list.

**Returns** Entries of the list, they need to contain a locale-independent id and a localized content string, for example: ["ENTRY_1_ID", "entry 1"], [ENTRY_2_ID", "entry 2"]

**get_dependent_components** () → Dict[str, List[str]]

Obtains a dictionary containing all UI elements currently connected regarding their visibility.

**Returns** All UI elements currently connected (key: dependent component id, value: required list entries to set it visible)

**get_selected_entry** () → str

Obtains the initially selected entry.

**Returns** Initially selected entry. If empty, the first element is selected.

**remove_component_visibility** *(dependent_component_id: str)*

Removes the visibility connection to a UI component that has been previously connected.

**Parameters** **dependent_component_id** – Id of the previously connection UI element

**set_index** *(index: int = -1)*

Sets the position in the graphical user interface.

**Parameters** **index** – The position

**Returns** This UI component
class `tc_python.propertymodel_sdk.UIPhaseListComponent` (component_id: str, name: str, description: str, default_phase: str = '', any_marker_setting: bool = False)

Bases: `tc_python.propertymodel_sdk.UIComponent`

Phase list UI component of the model panel.

Parameters

- **component_id** – Unique id of the component
- **name** – Name of the component, will be presented in the model panel
- **description** – Additional description of the component
- **default_phase** – Default phase, if omitted no default phase is chosen and only initially the first element of the list is selected. If an ANY-marker is added, this is chosen as the default element.
- **any_marker_setting** – Defines if an entry “ANY PHASE” should be added to the phase list, if set to true this overrides any default phase setting

`enable_add_button()`

Adds a + button to the right of the UI component.

**Returns**
This UI component

`enable_remove_button()`

Adds a - button to the right of the UI component.

** Returns**
This UI component

`get_any_marker_setting()` → bool

Obtains the setting if any entry “ANY PHASE” is added to the phase list.

**Returns**
If an entry “ANY PHASE” is added to the phase list, if set to true this overrides any default phase setting

`get_default_phase()` → str

Obtains the default phase.

**Returns**
Default phase, if omitted no default phase is chosen and only initially the first element of the list is selected. If an ANY-marker is added, this is chosen as the default element.

`set_index(index: int = - 1)`

Sets the position in the graphical user interface.

**Parameters**

- **index** – The position

**Returns**
This UI component

class `tc_python.propertymodel_sdk.UIMyStringComponent` (component_id: str, name: str, description: str, string: str)

Bases: `tc_python.propertymodel_sdk.UIComponent`

General text field UI component of the model panel.

Parameters

- **component_id** – Unique id of the component
- **name** – Name of the component, will be presented in the model panel
- **description** – Additional description of the component
• **string** – Initial setting of the text field

**enable_add_button()**
Adds a + button to the right of the UI component.

**Returns** This UI component

**enable_remove_button()**
Adds a - button to the right of the UI component.

**Returns** This UI component

**get_value() → str**
Obtains the setting of the text field.

**Returns** The setting of the text field

**set_index(index: int = -1)**
Sets the position in the graphical user interface.

**Parameters**

- **index** – The position

**Returns** This UI component

```python
class tc_python.propertymodel_sdk.UITemperatureComponent(component_id: str, 
name: str, description: str, temp: float)
```

**Bases:** `tc_python.propertymodel_sdk.UIComponent`

Temperature value text field UI component of the model panel.

**Parameters**

- **component_id** – Unique id of the component
- **name** – Name of the component, will be presented in the model panel
- **description** – Additional description of the component
- **temp** – Initial temperature to be set in the text field (unit defined by the user in the Thermo-Calc system)

**enable_add_button()**
Adds a + button to the right of the UI component.

**Returns** This UI component

**enable_remove_button()**
Adds a - button to the right of the UI component.

**Returns** This UI component

**get_temp() → float**
Obtains the temperature set in the text field.

**Returns** The temperature to be set in the text field (unit defined by the user in the Thermo-Calc system)

**set_index(index: int = -1)**
Sets the position in the graphical user interface.

**Parameters**

- **index** – The position

**Returns** This UI component
tc_python.propertymodel_sdk.create_boolean_ui_component(component_id: str, name: str, description: str, initial_setting: bool) →
tc_python.propertymodel_sdk.UIBooleanComponent

Creates a UI checkbox component for a boolean value. The value of that component can later be accessed during
the model evaluation.

Parameters

- **component_id** – Unique id of the component
- **name** – Name of the component, will be presented in the model panel
- **description** – Additional description of the component
- **initial_setting** – Initial setting of the checkbox

Returns The created component

tc_python.propertymodel_sdk.create_condition_list_ui_component(component_id: str, name: str, description: str)
→
tc_python.propertymodel_sdk.UIConditionListComponent

Creates a UI list component for all conditions defined in the system. The value of that component can later be
accessed during the model evaluation.

Parameters

- **component_id** – Unique id of the component
- **name** – Name of the component, will be presented in the model panel
- **description** – Additional description of the component

Returns The created component

tc_python.propertymodel_sdk.create_energy_quantity(quantity_id: str, description: str)
→
tc_python.propertymodel_sdk.ResultQuantity

Creates a UI energy result quantity (in J). When the model is evaluated, a value can be added to the quantity and
it will be used to transfer the result to the Thermo-Calc plot engine.

Parameters

- **quantity_id** – Unique id of the result quantity
- **description** – Additional description of the result quantity

Returns The created result quantity

tc_python.propertymodel_sdk.create_float_ui_component(component_id: str, name: str, description: str, value: float)
→
tc_python.propertymodel_sdk.UIFloatComponent

Creates a UI text field component for a real number. The value of that component can later be accessed during
the model evaluation.

Parameters

- **component_id** – Unique id of the component
- **name** – Name of the component, will be presented in the model panel
- **description** – Additional description of the component
• **value** – Initial setting of the text field

Returns The created component

tc_python.propertymodel_sdk.create_general_quantity(quantity_id: str, description: str) →
tc_python.propertymodel_sdk.ResultQuantity

Creates a general result quantity that can contain any type of result (without a unit). When the model is evaluated, a value can be added to the quantity and it will be used to transfer the result to the Thermo-Calc plot engine.

Parameters

• **quantity_id** – Unique id of the result quantity

• **description** – Additional description of the result quantity

Returns The created result quantity

tc_python.propertymodel_sdk.create_length_quantity(quantity_id: str, description: str) →
tc_python.propertymodel_sdk.ResultQuantity

Creates a length result quantity. When the model is evaluated, a value can be added to the quantity and it will be used to transfer the result to the Thermo-Calc plot engine.

Parameters

• **quantity_id** – Unique id of the result quantity

• **description** – Additional description of the result quantity

Returns The created result quantity

tc_python.propertymodel_sdk.create_list_ui_component(component_id: str, name: str, description: str, entry_list: List[Tuple[str, str]], selected_entry: str = '') →
tc_python.propertymodel_sdk.UIGeneralListComponent

Creates a UI list component for string entries. The value of that component can later be accessed during the model evaluation.

Parameters

• **component_id** – Unique id of the component

• **name** – Name of the component, will be presented in the model panel

• **description** – Additional description of the component

• **entry_list** – Entries of the list, they need to contain a locale-independent id and a localized content string, for example: [(“ENTRY_1_ID”, “entry 1”), (ENTRY_2_ID”, “entry 2”)]

• **selected_entry** – Entry to be initially selected. If omitted, by default the first element is selected.

Returns The created component

tc_python.propertymodel_sdk.create_phase_list_ui_component(component_id: str, name: str, description: str, default_phase: str = '', any_marker: bool = False) →
tc_python.propertymodel_sdk.UIPhaseListComponent

Creates a UI list component for all phases defined in the system. It is possible to select a default phase that is
supposed to be the expected phase selection for that list. The value of that component can later be accessed during the model evaluation.

A default phase is the phase that is initially selected and re-selected as soon as a currently selected phase is removed. If the default phase is not available, a “NONE”-marker will be created and used instead of the default phase. A typical use case for the default phase setting is a phase list that expects to contain the LIQUID-phase of a system.

Parameters

- **component_id** – Unique id of the component
- **name** – Name of the component, will be presented in the model panel
- **description** – Additional description of the component
- **default_phase** – Default phase, if omitted no default phase is chosen and only initially the first element of the list is selected. If an ANY-marker is added, this is chosen as the default element.
- **any_marker** – Defines if an entry “ANY PHASE” should be added to the phase list, if set to true this overrides any default phase setting

Returns The created component

```
tc_python.propertymodel_sdk.create_strength_quantity(quantity_id: str, description: str) -> tc_python.propertymodel_sdk.ResultQuantity
```

Creates a strength result quantity. When the model is evaluated, a value can be added to the quantity and it will be used to transfer the result to the Thermo-Calc plot engine.

Parameters

- **quantity_id** – Unique id of the result quantity
- **description** – Additional description of the result quantity

Returns The created result quantity

```
tc_python.propertymodel_sdk.create_string_ui_component(component_id: str, name: str, description: str, string: str) -> tc_python.propertymodel_sdk.UIStringComponent
```

Creates a UI text field component. The value of that component can later be accessed during the model evaluation.

Parameters

- **component_id** – Unique id of the component
- **name** – Name of the component, will be presented in the model panel
- **description** – Additional description of the component
- **string** – Initial setting of the text field

Returns The created component

```
tc_python.propertymodel_sdk.create_surface_energy_quantity(quantity_id: str, description: str) -> tc_python.propertymodel_sdk.ResultQuantity
```

Creates an energy result quantity (in J). When the model is evaluated, a value can be added to the quantity and it will be used to transfer the result to the Thermo-Calc plot engine.

Parameters
• **quantity_id** – Unique id of the result quantity

• **description** – Additional description of the result quantity

**Returns** The created result quantity

`tc_python.propertymodel_sdk.create_temperature_quantity(quantity_id: str, description: str) → tc_python.propertymodel_sdk.ResultQuantity`

Creates a temperature result quantity (in K). When the model is evaluated, a value can be added to the quantity and it will be used to transfer the result to the Thermo-Calc plot engine.

**Parameters**

• **quantity_id** – Unique id of the result quantity

• **description** – Additional description of the result quantity

**Returns** The created result quantity

`tc_python.propertymodel_sdk.create_temperature_ui_component(component_id: str, name: str, description: str, initial_temp: float) → tc_python.propertymodel_sdk.UITemperatureComponent`

Creates a UI text field component for a temperature value. The value of that component can later be accessed during the model evaluation.

**Parameters**

• **component_id** – Unique id of the component

• **name** – Name of the component, will be presented in the model panel

• **description** – Additional description of the component

• **initial_temp** – Initial temperature to be set in the text field. (The unit of initial_temp is Kelvin. The value in the text field will be automatically converted using the unit chosen by the user.)

**Returns** The created component

`tc_python.propertymodel_sdk.create_time_quantity(quantity_id: str, description: str) → tc_python.propertymodel_sdk.ResultQuantity`

Creates a time result quantity (in s). When the model is evaluated, a value can be added to the quantity and it will be used to transfer the result to the Thermo-Calc plot engine.

**Parameters**

• **quantity_id** – Unique id of the result quantity

• **description** – Additional description of the result quantity

**Returns** The created result quantity
5.8 Module "exceptions"

```python
exception tc_python.exceptions.APIErrorException
    Bases: tc_python.exceptions.GeneralException
    An exception that occurred during the communication with the API-server. It is normally not related to an error in the user program.

exception tc_python.exceptions.CalculationException
    Bases: tc_python.exceptions.TCException
    An exception that occurred during a calculation.

exception tc_python.exceptions.ComponentNotExistingException
    Bases: tc_python.exceptions.GeneralException
    The selected component is not existing.

exception tc_python.exceptions.DatabaseException
    Bases: tc_python.exceptions.CalculationException
    Error loading a thermodynamic or kinetic database, typically due to a misspelled database name or a database missing in the system.

exception tc_python.exceptions.DegreesOfFreedomNotZeroException
    Bases: tc_python.exceptions.CalculationException
    The degrees of freedom in the system are not zero, i.e. not all required conditions have been defined. Please check the conditions given in the exception message.

exception tc_python.exceptions.EquilibriumException
    Bases: tc_python.exceptions.CalculationException
    An equilibrium calculation has failed, this might happen due to inappropriate conditions or a very difficult problem that can not be solved.

exception tc_python.exceptions.GeneralCalculationException
    Bases: tc_python.exceptions.CalculationException
    General error occurring while a calculation is performed.

exception tc_python.exceptions.GeneralException
    Bases: tc_python.exceptions.TCException
    A general exception that might occur in different situations.

exception tc_python.exceptions.InvalidCalculationConfigurationException
    Bases: tc_python.exceptions.CalculationException
    Thrown when errors are detected in the configuration of the calculation.

exception tc_python.exceptions.InvalidCalculationStateException
    Bases: tc_python.exceptions.CalculationException
    Trying to access an invalid calculation object that was invalidated by calling `invalidate` on it.

exception tc_python.exceptions.InvalidNumberOfResultGroupsException
    Bases: tc_python.exceptions.ResultException
    A calculation result contains several result groups, which is not supported for the used method.

exception tc_python.exceptions.InvalidResultConfigurationException
    Bases: tc_python.exceptions.ResultException
    A calculation result configuration is invalid.
```
exception tc_python.exceptions.InvalidResultStateException
   Bases: tc_python.exceptions.CalculationException
   Trying to access an invalid result (for example a SingleEquilibriumTempResult object that got already invalidated by condition changes or a result that was invalidated by calling invalidate on it).

exception tc_python.exceptions.LicenseException
   Bases: tc_python.exceptions.GeneralException
   No valid license for the API or any Thermo-Calc product used by it found.

exception tc_python.exceptions.NoDataForPhaseException
   Bases: tc_python.exceptions.ResultException
   There is no result data available for a selected phase.

exception tc_python.exceptions.NotAllowedOperationException
   Bases: tc_python.exceptions.CalculationException
   The called method or operation is not allowed in the current mode of operation (i.e. debug or production mode). Production mode means that the Property Model is only present as an *.py.encrypted-file, while in debug mode it is available as *.py-file. Certain methods for obtaining internal model parameters are not available for encrypted models.

exception tc_python.exceptions.PhaseNotExistingException
   Bases: tc_python.exceptions.GeneralException
   The selected phase is not existing, so no data can be provided for it.

exception tc_python.exceptions.ResultException
   Bases: tc_python.exceptions.TCException
   An exception that occurred during the configuration of a calculation result.

exception tc_python.exceptions.SyntaxException
   Bases: tc_python.exceptions.CalculationException
   Syntax error in a Console Mode expression.

exception tc_python.exceptions.TCException
   Bases: Exception
   The root exception of TC-Python.

exception tc_python.exceptions.UnrecoverableCalculationException
   Bases: tc_python.exceptions.CalculationException
   The calculation reached a state where no further actions are possible, this happens most often due to a FORTRAN- hard crash in the API server backend.

Note: It is possible to catch that exception outside of the with-clause context and to continue by setting up a new context (i.e. by a new with TCPython() as session).

tc_python.exceptions.handle_exception(e)
5.9 Module “abstract_base”

```python
class tc_python.abstract_base.AbstractCalculation(calculator)
    Bases: object

Abstract base class for calculations.

get_configuration_as_string() → str
    Returns detailed information about the current state of the calculation object.

    Warning: The structure of the calculator objects is an implementation detail and might change between releases without notice. Therefore do not rely on the internal object structure.

get_system_data() → tc_python.abstract_base.SystemData
    Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using with_system_modifications().

    Note: Parameters can only be read from unencrypted (i.e. user) databases loaded as *.tdb-file.

    Returns The system data

invalidate()
    Invalidates the object and frees the disk space used by it. This is only required if the disk space occupied by the object needs to be released during the calculation. No data can be retrieved from the object afterwards.

with_system_modifications(system_modifications: tc_python.abstract_base.SystemModifications)
    Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

    Note: This is only possible if the system has been read from unencrypted (i.e. user) databases loaded as a *.tdb-file.

    Parameters system_modifications – The system modification to be performed

    Returns
```

```python
class tc_python.abstract_base.AbstractResult(result)
    Bases: object

Abstract base class for results. This can be used to query for specific values.

invalidate()
    Invalidates the object and frees the disk space used by it. This is only required if the disk space occupied by the object needs to be released during the calculation. No data can be retrieved from the object afterwards.

class tc_python.abstract_base.PhaseParameter(parameter_name: Union[str, object])
    Bases: object

Database phase parameter expression used by SystemModifications.set().

    Parameters parameter_name – The phase parameter name

get_intervals() → List[tc_python.abstract_base.TemperatureInterval]
    Returns the list of all defined intervals.
```
Returns The defined temperature intervals

get_lower_temperature_limit() → float

Returns the lower temperature limit.

get_name() → str

Returns the name of the phase parameter.

remove_all_intervals()

Removes all previously defined temperature intervals.

remove_interval_with_upper_limit(upper_temperature_limit: float)

Removes a previously defined temperature interval with matching upper temperature limit. If no such interval exists, an exception is thrown.

set_expression_with_upper_limit(parameter_expression: str, upper_temperature_limit: float = 6000.0)

_adds/overwrites a parameter expression for a temperature interval.

Default value of the upper limit of the interval: 6000 K

Notes: The lower temperature limit is either defined by the lower temperature limit given with PhaseParameter.set_lower_temperature_limit() or by the upper temperature limit of the adjacent interval.

Notes: If there is an existing interval with exactly the same upper_temperature_limit, that interval is overwritten, otherwise the interval is added.

Parameters

• parameter_expression – The parameter expression, example:
  +V34*T*LN(T)+V35*T**2+V36*T**(-1)+V37*T**3")

• upper_temperature_limit – The upper temperature limit for which the expression should be used

Returns This PhaseParameter object

set_interval(interval: tc_python.abstract_base.TemperatureInterval)

Adds/overwrites a temperature interval.

Notes: The lower temperature limit is either defined by the lower temperature limit given with PhaseParameter.set_lower_temperature_limit() or by the upper temperature limit of the adjacent interval.
Note: If there is an existing interval with exactly the same `upper_temperature_limit`, that interval is overwritten, otherwise the interval is added.

**Returns** This `PhaseParameter` object

`set_lower_temperature_limit(lower_temperature_limit: float = 298.15)`
Sets the lower temperature limit of the phase parameter.

**Default**: 298.15 K

**Parameters** `lower_temperature_limit` – The lower temperature limit in K

**Returns** This `PhaseParameter` object

```python
class tc_python.abstract_base.SystemData(system_data)
Bases: object

Provides information about the parameters and functions of a user database. The obtained objects can be used to modify the database using `with_system_modifications()` of all calculators.
```

**Note:** Parameters can only be read from unencrypted (i.e. `user`) databases loaded as *.tdb*-file.

`get_phase_parameter(parameter: str) → tc_python.abstract_base.PhaseParameter`
Returns a phase parameter.

**Example:**
```
system_data.get_phase_parameter('G(HCP_A3,FE:VA;0)')
```

**Note:** Parameters can only be read from unencrypted (i.e. `user`) databases loaded as a *.tdb*-file.

```
Note: For details about the syntax search the Thermo-Calc help for GES (the name for the Gibbs Energy System module in Console Mode).
```

**Parameters** `parameter` – The name of the phase parameter (for example: “G(LIQUID,FE;0)"

**Returns** The phase parameter

`get_phase_parameter_names() → List[str]`
Returns all phase parameters present in the current system.

**Returns** The list of phase parameters

`get_system_function(f: str) → tc_python.abstract_base.SystemFunction`
Returns a system function.

**Note:** The parameter ‘f’ was previously called ‘function’ but was renamed.

**Example:**
```
system_data.get_system_function('GHSERCR')
```
Note: Functions can only be read from unencrypted (i.e. user) databases loaded as a *.tdb-file.

Note: For details about the syntax search the Thermo-Calc help for GES (the name for the Gibbs Energy System module in Console Mode).

**Parameters** f – The name of the system function (for example: “GHSERCR”)

**Returns** The system function

`get_system_function_names()` → List[str]

Returns all system functions present in the current system.

**Returns** The list of system functions

```python
class tc_python.abstract_base.SystemFunction(function_name: Union[str, object])
Bases: object
```

Database function expression used by `SystemModifications.set()`.

**Parameters** function_name – The function name

`get_intervals()` → List[tc_python.abstract_base.TemperatureInterval]

Returns the list of all defined intervals.

**Returns** The defined temperature intervals

`get_lower_temperature_limit()` → float

Returns the lower temperature limit.

**Returns** The lower temperature limit in K

`get_name()` → str

Returns the name of the system function.

**Returns** The name of the system function

`remove_all_intervals()`

Removes all previously defined temperature intervals.

**Returns** This `SystemFunction` object

`remove_interval_with_upper_limit(upper_temperature_limit: float)`

Removes a previously defined temperature interval with matching upper temperature limit.

If no such interval exists, an exception is thrown.

**Returns** This `SystemFunction` object

`set_expression_with_upper_limit(function_expression: str, upper_temperature_limit: float = 6000.0)`

Adds/overwrites a function expression for a temperature interval.

Default value of the upper limit of the interval: 6000 K

**Note:** The lower temperature limit is either defined by the lower temperature limit given with `SystemFunction.set_lower_temperature_limit()` or by the upper temperature limit of the adjacent interval.
**Note:** If there is an existing interval with exactly the same `upper_temperature_limit`, that interval is overwritten, otherwise the interval is added.

**Parameters**

- **function_expression** – The function expression, example:
  
  
  \[ +V34*T*LN(T)+V35*T**2+V36*T**(-1)+V37*T**3 \]

- **upper_temperature_limit** – The upper temperature limit for which the expression should be used

**Returns** This `SystemFunction` object

```python
set_interval(interval: tc_python.abstract_base.TemperatureInterval)
```

Adds/overwrites a temperature interval.

**Note:** The lower temperature limit is either defined by the lower temperature limit given with `SystemFunction.set_lower_temperature_limit()` or by the upper temperature limit of the adjacent interval.

**Note:** If there is an existing interval with exactly the same `upper_temperature_limit`, that interval is overwritten, otherwise the interval is added.

**Returns** This `SystemFunction` object

```python
set_lower_temperature_limit(lower_temperature_limit: float = 298.15)
```

Sets the lower temperature limit of the system function.

**Default:** 298.15 K

**Parameters** `lower_temperature_limit` – The lower limit in K

**Returns** This `SystemFunction` object

```python
class tc_python.abstract_base.SystemModifications
```

Functionality to modify a user database during a calculation by changing phase parameters and system functions.

The actual changes are only applied by using `tc_python.abstract_base.AbstractCalculation.with_system_modifications()` on a calculator object.

```python
run_ges_command(ges_command: str)
```

Sends a GES-command. This is actually applied when running `with_system_modifications` on a calculator object.

Example: `run_ges_command("AM-PH-DE FCC_A1 C_S 2 Fe:C")` for adding a second composition set to the FCC_A1 phase with Fe as major constituent on first sublattice and C as major constituent on second sublattice.

**Note:** For details about the syntax search the Thermo-Calc help for GES (the name for the Gibbs Energy System module in Console Mode).
Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw GES-commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten parenthesis, ...).

Parameters **ges_command** – The GES-command (for example: “AM-PH-DE FCC_A1 C_S 2 Fe:C”)

Returns This `SystemModifications` object

```python
def set(parameter_or_function: Union[tc_python.abstract_base.PhaseParameter, tc_python.abstract_base.SystemFunction])
    Overwrites or creates a phase parameter or system function.
```

Example: `system_modifications.set(PhaseParameter('G(LIQUID,FE;0)').set_expression_with_upper_limit('+1.2*GFELIQ'))`

Example: `system_modifications.set(SystemFunction("DGDEF").set_expression_with_upper_limit('+10.0-R*T', 1000).set_expression_with_upper_limit('+20.0-R*T', 3000))`

Note: The old parameter/function is **overwritten** and any temperature intervals not defined are lost.

Note: Please consult the Thermo-Calc GES-system documentation for details about the syntax.

Returns This `SystemModifications` object

```python
class tc_python.abstract_base.TemperatureInterval(expression: Union[str, object], upper_temperature_limit: float)
```

Bases: `object`

Temperature interval expression used by the classes `SystemFunction` and `PhaseParameter`.

Parameters

- **expression** – The temperature function expressed in Thermo-Calc database syntax.
- **upper_temperature_limit** – The upper temperature limit in K

```python
def get_expression() → str
    Returns the function expression of this temperature interval.
```

Returns The temperature function expression

```python
def get_upper_temperature_limit() → float
    Returns the upper limit of this temperature interval.
```

Returns The upper temperature limit in K

```python
def set_expression(expression: str)
    Sets the function expression of this temperature interval.
```

Parameters **expression** – The temperature function expression

```python
def set_upper_temperature_limit(upper_temperature_limit: float)
    Sets the upper limit of this temperature interval.
```

Returns The upper temperature limit in K
Parameters `upper_temperature_limit` – The upper temperature limit in K
This section provides an FAQ for common problems that occur when using TC-Python.

### 6.1 Diagnostics script

If you have problems running TC-Python, run the diagnostics script below.

On Linux you can alternatively download the script directly into your current working directory by:

```
$ curl -O https://www2.thermocalc.com/downloads/support/diagnostics-py/tc-python-
   →diagnostic-script-2022b.py
```

```python
#***
# Run this script when troubleshooting TC-Python

# It is important to run this script EXACTLY the same way as you run your TC-Python
# script
# (In the same IDE, same project, same Python environment, same Jupyter notebook etc)
#***

version = '2022b'

print('Testing TC-Python version: ' + version)
print('Please make sure that the variable "version" above, matches the release that
you want to test, if not change it and re-run this script."

# below this line, nothing needs to be manually updated.

import sys

print('Python version: (should be at least 3.5 and can NOT be older than 3.0)')
print(str(sys.version_info[0]) + '.' + str(sys.version_info[1]))
if sys.version_info[0] < 3 or sys.version_info[1] < 5:
    print('Wrong version of Python !!!!!')

print('Python executable path: (gives a hint about the used virtual / conda
   environment, in case of Anaconda the corresponding 
   environment name can be found by running `conda env list` on the Anaconda
   command prompt, ')
   
   'TC-Python must be installed into \nEACH separate environment used!')
print(sys.executable)
```

(continues on next page)
import os
print('')
print('Thermo-Calc ' + version + ' installation directory: (must be a valid path to a complete installation of ' + version + ')')
tc_env_variable = 'TC' + version[2:].upper() + '_HOME'
try:
    print(os.environ[tc_env_variable])
except:
    print('No Thermo-calc environment variable for ' + version + ' was found. (' + tc_env_variable + ')')

print('')
print('Url of license server: (if license server is NO-NET, you need a local license file)')
try:
    print(os.environ['LSHOST'])
except:
    print('No Thermo-calc license server url was found. (LSHOST)')

print('')
print('Path to local license file: (only necessary if not using license server)')
try:
    print(os.environ['LSERVRC'])
except:
    print('No path to local license file was found. (LSERVRC)')

import tc_python
numerical_version = version[:-1]
if version[-1] == 'a':
    numerical_version += '.1.*'
elif version[-1] == 'b':
    numerical_version += '.2.*'
print('')
print('TC-Python version: (needs to be ' + numerical_version + ')')
print(tc_python.__version__)

with tc_python.TCPython() as session:
    print('')
    print('Lists the databases: (should be a complete list of the installed databases, that you have license for or do not require license)')
    print(session.get_databases())
6.2 “No module named tc_python” error on first usage

This problem occurs because your used Python interpreter cannot find the TC-Python package. We expect that you have installed the TC-Python package in your Python system interpreter following the instructions in the Installation Guide.

Normally the error message “No module named tc_python” is caused by unintentionally configuring a PyCharm project to use a so-called Virtual Environment. This happens unfortunately by default when creating a new PyCharm project with not changing the default settings.

**Note:** A Virtual Environment is basically a separate and completely independent copy of the system-wide Python interpreter. It does not contain any packages.

On Windows systems we recommend to use the Anaconda Python Distribution as Python interpreter. However, the instructions given here are valid for any operating system and distribution.

Since TC-Python 2018b we do recommend to **not use Virtual Environments** unless there is a reasonable use case for that.

There are two possible solutions to fix the problem:

1. **The quick fix for your problem is to run**

   ```bash
   pip install <path to the TC-Python folder>/TC_Python-<version>-py3-none-any.whl
   ```

   within the Terminal window of the opened PyCharm project. This Terminal window automatically runs within the Virtual Environment configured for the project (if any). You can see the name of the Virtual Environment at the beginning of each command prompt line (here it is called `venv`):

   ![Command Prompt Example](image)

   The command will consequently **install TC-Python also within that Virtual Environment automatically**. The Terminal window can be found at the bottom of the IDE. Note that it might be necessary to enable these buttons first by selecting the menu entry View → Tool Buttons.

2. **The better fix is to change your project to use the system interpreter.** This is described in detail in the section Fixing potential issues with the environment in Step 5 of the Installation Guide.

   It is recommendable to use that approach also for all your future projects.

Both fixes will only change the configuration of the opened project. Further useful information can be found in the section Python Virtual Environments.
6.3 “pip install” fails with “Failed to establish a new network connection” or similar

If `pip install` fails with a network related error (might also be “socket not available”, “retrying after connection broken”, ...) it is often due to the computer being behind a proxy-server, this is common in large organizations. Of course also the network connection might be broken.

TC-Python has dependencies to a few other packages:
- `py4j`
- `jproperties`
- `six` (transient dependency of `jproperties`)

1. The recommended approach is to simply use `pip`. It will resolve the dependencies automatically by downloading them from the PyPI-repository server (https://pypi.org). If your computer is located behind a proxy-server, the connection to the repository will fail. In that case it is necessary to configure `pip` with the detailed configuration of the proxy server:

```
pip install -proxy user:password@proxy_ip:port py4j jproperties
```

2. Another alternative is to manually download the latest *.whl-file of each dependency from the repository server (https://pypi.org -> Search projects) and to install it manually using:

```
pip install py4j-#.#.#-py2.py3-none-any.whl
...
```

The actual actual version number needs to be inserted into the file name. The downside of this approach is that updates to that package have to be fully manual also in the future. Additionally it is also necessary to install all transient dependencies in that way.
tc_python.abstract_base, 267
tc_python.batch_equilibrium, 52
tc_python.diffusion, 117
tc_python.entities, 224
tc_python.exceptions, 265
tc_python.material_to_material, 160
tc_python.precipitation, 57
tc_python.process_metallurgy.base, 183
tc_python.process_metallurgy.equilibrium, 188
tc_python.process_metallurgy.process, 198
tc_python.propertymodel, 155
tc_python.propertymodel_sdk, 248
tc_python.quantity_factory, 234
tc_python.scheil, 79
tc_python.server, 228
tc_python.single_equilibrium, 39
tc_python.step_or_map_diagrams, 89
tc_python.system, 216
tc_python.utils, 246