
TC-Python Documentation

Release 2019a

Thermo-Calc Software AB

Nov 23, 2018

CONTENTS

1	TC-Python Quick Install Guide	1
1.1	Step 1: Install a Python Distribution	1
1.1.1	Install Anaconda	1
1.2	Step 2: Install Thermo-Calc and the TC-Python SDK	1
1.3	Step 3: Install TC-Python	2
1.4	Step 4: Install an IDE (Integrated Development Environment)	2
1.5	Step 5: Open the IDE and Run a TC-Python Example	3
1.5.1	Open the TC-Python Project in PyCharm	3
1.5.2	Fixing potential issues with the environment	3
1.6	Updating to a newer version	4
2	Mac OS: Setting Environment Variables	5
3	Architecture overview	7
3.1	TCPython	7
3.2	SystemBuilder and System	8
3.3	Calculation	8
3.3.1	Single equilibrium calculations	9
3.3.2	Precipitation calculations	9
3.3.3	Scheil calculations	10
3.3.4	Property diagram calculations	10
3.3.5	Phase diagram calculations	11
3.3.6	Diffusion calculations	11
3.3.7	Property model calculations	12
3.4	Result	12
4	Best Practices	15
4.1	Re-use and saving of results	15
4.2	All TC-Python objects are non-copyable	15
4.3	Python Virtual Environments	16
4.4	Using <i>with TcPython()</i> efficiently	16
4.5	Parallel calculations	17
4.6	Handling crashes of the calculation engine	18
4.7	Using TC-Python within a Jupyter Notebook or the Python console	19
5	API Reference	21
5.1	Calculations	21
5.1.1	Module “single_equilibrium”	21
5.1.2	Module “precipitation”	28
5.1.3	Module “scheil”	45

5.1.4	Module “step_or_map_diagrams”	50
5.1.5	Module “diffusion”	69
5.1.6	Module “propertymodel”	95
5.2	Module “system”	99
5.3	Module “entities”	104
5.4	Module “server”	107
5.5	Module “quantity_factory”	111
5.6	Module “utils”	122
5.7	Module “exceptions”	124
5.8	Module “abstract_base”	125
6	Troubleshooting	127
6.1	“No module named tc_python” error on first usage	127
6.2	“pip install” fails with “Failed to establish a new network connection” or similar	128
	Python Module Index	129

TC-PYTHON QUICK INSTALL GUIDE

This quick 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 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.

1.1.1 Install Anaconda

1. Navigate to the Anaconda website: <https://www.anaconda.com/download/>.
2. Click to choose your OS (operating system) and then click **Download**. Follow the instructions. It is recommended you keep all the defaults.

1.2 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 and choose a **Custom** installation.

See Custom Standalone Installation in the *Thermo-Calc Installation Guide*.

If you have already installed Thermo-Calc, you need to find the installation file (e.g. Windows *.exe, Mac *.zip and Linux *.run) to relaunch the installer and then continue with the next steps.

2. On the **Select Components** window, click to select the **TC-Python** check box.
3. On the **Install TC-Python** window, click **Next**.

4. When the installation is complete, the TC-Python folder opens and includes the *.whl file needed for the next step. There is also an Examples folder with Python files you can use in the IDE to understand and work with TC-Python.

The installation location for this API is the same as for other SDKs and based on the OS. For details, see *Default Directory Locations* in the *Thermo-Calc Installation Guide*.

1.3 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
```

For example, on a Windows OS Standalone custom installation, when you install for all users, the path to the TC-Python folder is `C:\Users\Public\Documents\Thermo-Calc\2019a\SDK\TC-Python\`

Details for Mac and Linux installations are described in *Default Directory Locations* in the *Thermo-Calc Installation Guide*. Note that on Linux typically `pip3` is used.

3. Press <Enter>. When the process is completed, there is a confirmation that TC-Python is installed.

1.4 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.

1. Navigate to the PyCharm website: <https://www.jetbrains.com/pycharm/download>.
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*.

1.5 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.

1.5.1 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 as done in Step 4.

For example, on a Windows OS Standalone custom installation, when you install for all users, the path to the TC-Python folder is `C:\Users\Public\Documents\Thermo-Calc\2019a\SDK\TC-Python\`

Details for Mac and Linux installations are described in the *Default Directory Locations* section in the *Thermo-Calc Installation Guide*.

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**.

1.5.2 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 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 *Best Practices* chapter.

1.6 Updating to a newer version

When updating to a newer version of Thermo-Calc, **you always need to also install the latest version of TC-Python**. It is not sufficient to run the installer of Thermo-Calc. The procedure is generally identical to *Step 3*:

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

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

```
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. Note that on Linux usually *pip3* is used.

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.

```
TC19A_HOME=/Applications/Thermo-Calc-2019a.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      LSEVRVC=/Applications/Thermo-Calc-2019a.app/  
Contents/Resources/lsevrvc
```

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.

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

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:

```
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
```

Note: For your information this starts a process running a calculation server. 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

```
# 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()`.

```
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.

```
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:

```
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 **Result**, 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()`.

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

Example:

```
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")
    )
```

3.3.2 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:

```
from tc_python import *

with TCPython() as start:
```

(continues on next page)

(continued from previous page)

```

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_Al").
                add_precipitate_phase(PrecipitatePhase("AL3SC"))).
            calculate()
    )

```

3.3.3 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:

```

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())
    )

```

3.3.4 Property diagram calculations

For the property diagram (step) calculation, everything that you can configure in the *Equilibrium Calculator* when choosing *Property diagram* 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:

```

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)).
    )

```

(continues on next page)

(continued from previous page)

```

        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"))
    )

```

3.3.5 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"),
                                                        ThermodynamicQuantity.
↪ temperature())
    )

```

3.3.6 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:

```

from tc_python import *

```

(continues on next page)

(continued from previous page)

```

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)

```

3.3.7 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:

```

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")

```

3.4 Result

All calculations have a method called `calculate()` that starts the calculations and when finished, returns a **Result**.

The **Result** classes have 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 disc by the method `save_to_disc()`

Previously saved results can be loaded by the method `load_result_from_disk()` on the **Setup** class.

Example:

```
# 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.

Example:

```
# ...
# 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")
```


BEST PRACTICES

4.1 Re-use and saving of results

To re-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()`.

```
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:

```
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:

```
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)
```

4.2 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:

```
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 ...
```

4.3 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 recommend in most cases to *install TC-Python within your global interpreter*, for example by running the `pip install` command within your default Anaconda prompt.

4.4 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()
```

(continues on next page)

(continued from previous page)

```

# ...
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)

```

4.5 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())

```

(continues on next page)

(continued from previous page)

```

        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

```

4.6 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"]).
                    get_system().
                    with_isothermal_diffusion_calculation().
                    set_temperature(temperature).
                    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))

```

(continues on next page)

(continued from previous page)

```

        ).
        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...')
```

4.7 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.







Operating system	Temporary directory
Windows	C:\Users{UserName}\AppData\Local\Temp\TC_TMP4747588488953835507
MacOS	/var/folders/g7/7du81ti_b7mm84n184fn3k910001g/T/ TC_TMP4747588488953835507
Linux	/tmp/TC_TMP4747588488953835507

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:

Jupyter TC-Python with Jupyter Notebook Last Checkpoint: a few seconds ago (autosaved)  Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python [conda env:Anaconda3]

      Code

```
In [1]: from tc_python import *

In [2]: start_api_server()

In [3]: system = SetUp().select_database_and_elements("FEDEMO", ["Fe", "Ni", "Cr"]).get_system()
        calc = system.with_single_equilibrium_calculation()

In [4]: temp = 825 # in K
        ni_conc = 10.0 # in wt-%
        cr_conc = 8.0 # in wt-%

        calc. \
            set_condition(ThermodynamicQuantity.temperature(), temp). \
            set_condition(ThermodynamicQuantity.mass_fraction_of_a_component("Ni", ni_conc / 100). \
            set_condition(ThermodynamicQuantity.mass_fraction_of_a_component("Cr", cr_conc / 100)
        result = calc.calculate()

In [5]: result.get_value_of(ThermodynamicQuantity.mole_fraction_of_a_phase("FCC_A1"))

Out[5]: 0.3345580340424432

In [6]: stop_api_server()
```


5.1 Calculations

5.1.1 Module “single_equilibrium”

class `tc_python.single_equilibrium.SingleEquilibriumCalculation` (*calculator*)

Bases: `tc_python.abstract_base.AbstractCalculation`

Configuration for a single equilibrium calculation.

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

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 behaviour to use that object after the state of the calculation has been changed.

<p>Warning: If the result object should be valid for the whole program lifetime, use <code>calculate_with_state()</code> instead.</p>
--

calculate_with_state () → `tc_python.single_equilibrium.SingleEquilibriumResult`

Performs the calculation and provides a result object that will reflect the present state of the calculation during the whole lifetime of the object. This method comes with a performance and temporary disk space overhead. It should only be used if it is necessary to access the result object again later after the state has been changed. In most cases you should use the method `calculate()`.

Returns A new `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.

Returns This `SingleEquilibriumCalculation` object

enable_global_minimization ()

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

Returns This `SingleEquilibriumCalculation` 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 (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 or whatsoever.

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.

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 ThermodynamicQuantity to set as condition, a console syntax strings 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 POLY-module immediately in the engine.

Parameters command – The Thermo-Calc console 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.

<p>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).</p>
--

set_component_to_entered (*component: str = '*'*)

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

Parameters component – The component name

Returns This *SingleEquilibriumCalculation* object

set_component_to_suspended (*component: str = '*'*)

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

Parameters component – The component name

Returns This *SingleEquilibriumCalculation* object

set_condition (*quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], value: float*)

Sets the specified condition.

Parameters

- **quantity** – The ThermodynamicQuantity to set as condition, a console syntax string can be used as an alternative (for example “X(Cr)”)
- **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 or whatsoever.

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

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
- **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

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 = 'current_temperature', 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

class tc_python.single_equilibrium.**SingleEquilibriumOptions**

Bases: object

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 favourable 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 *SingleEquilibriumCalculation* 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 favourable 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 *SingleEquilibriumCalculation* object

set_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 *SingleEquilibriumCalculation* 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 *SingleEquilibriumCalculation* 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 *SingleEquilibriumCalculation* 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 *SingleEquilibriumCalculation* 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 `tc_python.single_equilibrium.SingleEquilibriumResult.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 `tc_python.single_equilibrium.SingleEquilibriumResult.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 *ThermodynamicQuantity* to get the value of, a console syntax strings can be used as an alternative (for example “NPM(FCC_A1)”)

Returns The requested value

save_to_disk (*path: str*)

Saves the result to disc. Note tha 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 *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 behaviour 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 `tc_python.single_equilibrium.SingleEquilibriumResult.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 `tc_python.single_equilibrium.SingleEquilibriumResult.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

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

get_conditions () → 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()` → 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()` → 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])` → float

Returns a value from a single equilibrium calculation.

Parameters **quantity** – the `ThermodynamicQuantity` to get the value of, a console 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

5.1.2 Module “precipitation”

`class tc_python.precipitation.GrowthRateModel`

Bases: `enum.Enum`

Choice of the used **growth rate model** for a precipitate.

ADVANCED = 3

The **ADVANCED MODEL** was 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.

Note: Since it is not always possible to solve the flux-balance equations and it takes time, and where possible, use a less rigorous but simple and efficient model is preferred.

GENERAL = 5

The **GENERAL MODEL** ...

SIMPLIFIED = 2

The **SIMPLIFIED MODEL** is based on the advanced model but avoids the difficulty to find the operating tie-line and uses 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 = 5000000000000.0*)
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.

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

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 m³/mol if the database contains no molar volume information).

Parameters volume – The molar volume [m³/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 [-]

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 radius 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 (*max_rel_change_nucleation_rate_log: 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 (*max_rel_radius_change: 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 (*max_rel_solute_composition_change: 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 (*max_time_step: 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 (*max_time_step_during_heating: 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 (*max_volume_fraction_dissolve_time_step: 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 (*min_radius_nucleus_as_particle: 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` (*min_radius_points_per_magnitude: 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` (*radius_points_per_magnitude: 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` (*rel_radius_change_class_collision: 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

`set_volume_fraction_of_phase_type` (*volume_fraction_of_phase_type_enum:*

`tc_python.precipitation.VolumeFractionOfPhaseType`)

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.

Note: This class is only relevant if the option `TransformationStrainCalculationOption.USER_DEFINED` has been chosen using `PrecipitatePhase.set_transformation_strain_calculation_opti`. The elastic strain can only be considered for non-spherical precipitates.

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

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`

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 ()

Will disable 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 ()

Will enable 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 gets automatically calculated using a broken-bond model.

Parameters `interfacial_energy` – The interfacial energy [J/m²]

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 m³/mol is used.

Parameters `volume` – The molar volume [m³/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 [degrees]
- **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 [degrees]
- **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_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 [degrees]
- **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_in_bulk (*number_density: float = -1*)

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) [m⁻³]

Returns This *PrecipitatePhase* object

set_phase_boundary_mobility (*phase_boundary_mobility: float*)

Sets the phase boundary mobility. **Default:** 10.0 m⁴/(Js).

Parameters **phase_boundary_mobility** – The phase boundary mobility [m⁴/(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` has been 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 () → `tc_python.precipitation.PrecipitationCalculationTTTorCCTResult`

Runs the CCT-diagram calculation.

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

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 `PrecipitationCalculation` 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 `PrecipitationCalculation` 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 PrecipitationCalculation object

with_numerical_parameters (*numerical_parameters: tc_python.precipitation.NumericalParameters*)

Sets the numerical parameters. If not specified, reasonable defaults will be used.

Parameters **numerical_parameters** – The parameters

Returns This PrecipitationCalculation object

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 tha 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 `PrecipitationCalculationResult` object

class `tc_python.precipitation.PrecipitationCalculationSingleResult` (*result*)

Bases: `tc_python.precipitation.PrecipitationCalculationResult`

Result of a isothermal or non-isothermal precipitation calculation. This can be used to query for specific values. A detailed definition of the axis variables can be found in the Help.

get_aspect_ratio_distribution_for_particle_length_of (*precipitate_id*: *str*,
time: *float*) → [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 PrecipitateMorphology.NEEDLE or PrecipitateMorphology.PLATE.

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 (mean particle length [m], aspect ratio)

get_aspect_ratio_distribution_for_radius_of (*precipitate_id*: *str*, *time*: *float*) → [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 PrecipitateMorphology.NEEDLE or PrecipitateMorphology.PLATE.

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 (mean radius [m], aspect ratio)

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

Returns the critical radius 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], critical radius [m])

get_cubic_factor_distribution_for_particle_length_of (*precipitate_id*: *str*,
time: *float*) → [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 PrecipitateMorphology.CUBOID.

Parameters

- **time** – The time in seconds
- **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*) → [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*) → [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_matrix_composition_in_mole_fraction_of (*element_name: str*) → [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: str*) → [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: str*) → [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: str*) → [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: str*) → [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: str*) → [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 phase name or alias

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

get_nucleation_rate_of (*precipitate_id: str*) → [typing.List[float], 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 [m⁻³ s⁻¹])

get_number_density_distribution_for_particle_length_of (*precipitate_id*: *str*,
time: *float*) →
[typing.List[float],
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 [m⁻⁴])

get_number_density_distribution_for_radius_of (*precipitate_id*: *str*, *time*: *float*)
→ [typing.List[float], 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 [m⁻⁴])

get_number_density_of (*precipitate_id*: *str*) → [typing.List[float], 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 [m⁻³])

get_precipitate_composition_in_mole_fraction_of (*precipitate_id*: *str*, *element_name*: *str*) → [typing.List[float], 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.PrecipitationCalculationResult`

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 () → `tc_python.precipitation.PrecipitationCalculationSingleResult`

Runs the isothermal precipitation calculation.

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

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 `PrecipitationCalculation` 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 `PrecipitationCalculation` 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 `PrecipitationCalculation` object

with_numerical_parameters (*numerical_parameters: tc_python.precipitation.NumericalParameters*)

Sets the numerical parameters. If not specified, reasonable defaults will be used.

Parameters `numerical_parameters` – The parameters

Returns This `PrecipitationCalculation` object

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

Bases: `tc_python.abstract_base.AbstractCalculation`

Configuration for a non-isothermal precipitation calculation.

calculate () → `tc_python.precipitation.PrecipitationCalculationSingleResult`

Runs the non-isothermal precipitation calculation.

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

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 `PrecipitationCalculation` 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 `PrecipitationCalculation` 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 PrecipitationCalculation object

with_numerical_parameters (*numerical_parameters: tc_python.precipitation.NumericalParameters*)

Sets the numerical parameters. If not specified, reasonable defaults will be used.

Parameters **numerical_parameters** – The parameters

Returns This PrecipitationCalculation 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 () → `tc_python.precipitation.PrecipitationCalculationTTTOrCCTResult`

Runs the TTT-diagram calculation.

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

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 PrecipitationCalculation 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 PrecipitationCalculation object

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

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

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

set_temperature_step (*temperature_step: float*)

Sets the temperature step for the TTT-diagram, if unset the default value is 10 K.

Parameters **temperature_step** – the temperature step [K]

Returns This PrecipitationTTTCalculation object

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 PrecipitationCalculation object

with_numerical_parameters (*numerical_parameters: tc_python.precipitation.NumericalParameters*)

Sets the numerical parameters. If not specified, reasonable defaults will be used.

Parameters **numerical_parameters** – The parameters

Returns This PrecipitationCalculation object

class `tc_python.precipitation.TransformationStrainCalculationOption`

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`

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%).

5.1.3 Module “scheil”

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()`.

calculate () → `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

Returns A `ScheilCalculationResult` which later can be used to get specific values from the simulation.

disable_global_minimization ()

Disables global minimization.

Default: Disabled

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

enable_global_minimization ()

Enables global minimization.

Default: Disabled

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

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: 1>*)

Sets the composition unit.

Default: Mole percent (`CompositionUnit.MOLE_PERCENT`).

Parameters **unit_enum** – The new composition unit

Returns This *ScheilCalculation* 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 *ScheilCalculation* 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 *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

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 *ScheilCalculation* 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 *ScheilCalculation* 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 *ScheilCalculation* object

with_options (*options: tc_python.scheil.ScheilOptions*)

Sets the simulation options.

Parameters `options` – The simulation options

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 in 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 syntax strings can be used as an alternative (for example “T”)

- **y_quantity** – The second Scheil quantity (“y-axis”), console 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 Dict 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 syntax strings can be used as an alternative (for example “T”)
- **y_quantity** – The second Scheil quantity (“y-axis”), console 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 Dict containing the `ResultValueGroup` dataset objects with their “*stable phases*” labels as keys

get_values_of (*x_quantity*: `Union[tc_python.quantity_factory.ScheilQuantity, str]`, *y_quantity*: `Union[tc_python.quantity_factory.ScheilQuantity, 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 `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 syntax strings can be used as an alternative (for example “T”)
- **y_quantity** – The second Scheil quantity (“y-axis”), console syntax strings can be used as an alternative (for example “NV”)

Returns A tuple containing the x- and y-data in lists

save_to_disk (*path: str*)

Saves the result to disc. Note tha 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.

Returns this *ScheilCalculationResult* object

class `tc_python.scheil.ScheilOptions`

Bases: `object`

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 favourable 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 ()

disable_force_positive_definite_phase_hessian ()

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 favourable 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 ()

enable_force_positive_definite_phase_hessian ()

set_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_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

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

5.1.4 Module “step_or_map_diagrams”

class `tc_python.step_or_map_diagrams.AbstractAxisType`

Bases: `object`

The abstract base class for all axis types.

get_type () → `str`

Convenience method for getting the axis type.

Returns The type

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*

Bases: *enum.Enum*

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

class *tc_python.step_or_map_diagrams.Linear*

Bases: *tc_python.step_or_map_diagrams.AbstractAxisType*

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.AbstractAxisType`

Represents a logarithmic axis.

Note: A logarithmic axis is useful for low fractions like in a gas phase where 1E-7 to 1E-2 might be an interesting range. For the pressure a logarithmic axis is often also useful.

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.abstract_base.AbstractCalculation`

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 POLY module 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*) → `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.

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 ()
Dont 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 (*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 or whatsoever.

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.

keep_default_equilibria ()
Keep the initial equilibria added by default. This is only relevant in combination with `add_initial_equilibrium`

Default behaviour 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 *PhaseDiagramCalculation* to set as condition, a console syntax strings 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 POLY-module immediately in the engine.

Parameters **command** – The Thermo-Calc console 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_component_to_entered (*component*: *str = '*'*)

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

Parameters **component** – The component name

Returns This *PhaseDiagramCalculation* object

set_component_to_suspended (*component*: *str = '*'*)

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

Parameters **component** – The component name

Returns This *PhaseDiagramCalculation* object

set_condition (*quantity*: *Union[tc_python.quantity_factory.ThermodynamicQuantity, str]*, *value*: *float*)

Sets the specified condition.

Parameters

- **quantity** – The *ThermodynamicQuantity* to set as condition, a console syntax string can be used as an alternative (for example X(Cr))
- **value** – The value of the condition

Returns This *PhaseDiagramCalculation* 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 or whatsoever.

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

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
- **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

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 = 'current_temperature', 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 (*axis: tc_python.step_or_map_diagrams.CalculationAxis*)

Sets the second calculation axis.

Parameters axis – The axis

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 favourable 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 ()

disable_force_positive_definite_phase_hessian ()

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 favourable 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 ()

enable_force_positive_definite_phase_hessian ()

set_global_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_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_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 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 will not be sorted (because they are unsortable due to their nature).

Parameters

- **x_quantity** – The first quantity (“x-axis”), console syntax strings can be used as an alternative (for example “T”)
- **y_quantity** – The second quantity (“y-axis”), console syntax strings can be used as an alternative (for example “NV”)

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 will not be sorted (because they are unsortable due to their nature).

Parameters

- **x_quantity** – The first quantity (“x-axis”), console syntax strings can be used as an alternative (for example “T”)
- **y_quantity** – The second quantity (“y-axis”), console syntax strings can be used as an alternative (for example “NV”)

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 tha 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

class `tc_python.step_or_map_diagrams.PhaseDiagramResultValues` (*phase_diagram_values_java*)
Bases: `object`

Represents the data of a phase diagram.

get_invariants () → `tc_python.utils.ResultValueGroup`

Returns the x- and y-datasets of all invariants in the phase diagram.

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 Dict 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` (*x: float, y: float, text: str*)
Bases: `object`

Represents a *phase label at a plot coordinate*, i.e. the stable phases that are present at that *plot* coordinate.

Variables

- **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]
- **text** – The label (i.e. the stable phases at that point in the phase diagram, for example “LIQUID + FCC_A1”)

class `tc_python.step_or_map_diagrams.PhaseNameStyle`
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.abstract_base.AbstractCalculation`

Configuration for a property diagram calculation.

Note: Specify the conditions, the calculation is performed with `calculate()`.

calculate (*keep_previous_results: bool = False*) → `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.

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 (*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 or whatsoever.

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.

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 *ThermodynamicQuantity* to set as condition, a console syntax strings 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 POLY-module immediately in the engine.

Parameters **command** – The Thermo-Calc console 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_component_to_entered (*component: str = '*'*)

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

Parameters **component** – The component name

Returns This *PropertyDiagramCalculation* object

set_component_to_suspended (*component: str = '*'*)

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

Parameters **component** – The component name

Returns This *PropertyDiagramCalculation* object

set_condition (*quantity: Union[tc_python.quantity_factory.ThermodynamicQuantity, str], value: float*)

Sets the specified condition.

Parameters

- **quantity** – The ThermodynamicQuantity to set as condition, a console 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 or whatsoever.

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

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
- **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

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 = 'current_temperature'*, *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

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 favourable 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()`

`disable_force_positive_definite_phase_hessian()`

`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 favourable 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()`

`enable_force_positive_definite_phase_hessian()`

`set_global_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_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_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) → 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 syntax strings can be used as an alternative (for example “T”)
- **y_quantity** – The second quantity (“y-axis”), console 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 Dict 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 syntax strings can be used as an alternative (for example “T”)
- **y_quantity** – The second quantity (“y-axis”), console 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 Dict 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")`.

Parameters

- **x_quantity** – The first Thermodynamic quantity (“x-axis”), console syntax strings can be used as an alternative (for example “T”)
- **y_quantity** – The second Thermodynamic quantity (“y-axis”), console syntax strings can be used as an alternative (for example “NV”)

Returns A tuple containing the x- and y-data in lists

save_to_disk (*path: str*)

Saves the result to disc. Note tha 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

5.1.5 Module “diffusion”

class `tc_python.diffusion.AbstractBoundaryCondition`

Bases: `object`

The abstract base class for all boundary conditions.

get_type () → `str`

Convenience method for getting the boundary condition type.

Returns The type

class `tc_python.diffusion.AbstractCalculatedGrid`

Bases: `tc_python.diffusion.AbstractGrid`

get_type () → `str`

Convenience method for getting the grid type.

Returns The type

class `tc_python.diffusion.AbstractElementProfile`

Bases: `object`

The abstract base class for all initial composition profile types.

get_type () → str

Convenience method for getting the initial concentration profile type.

Returns The type

class `tc_python.diffusion.AbstractGrid`

Bases: `object`

The abstract base class for all grids.

get_type () → str

Convenience method for getting the grid type.

Returns The type

class `tc_python.diffusion.AbstractSolver`

Bases: `object`

Abstract base class for the solvers (Classic, Homogenization and Automatic).

get_type () → str

Convenience method for getting the boundary condition type.

Returns The type

class `tc_python.diffusionAutomaticSolver`

Bases: `tc_python.diffusion.AbstractSolver`

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 () → str

The type of the solver.

Returns The type

set_flux_balance_equation_accuracy (*accuracy*: float = 1e-16)

Only valid if the classic solver 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 classic solver 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 classic solver 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 `closed_system()`

Returns a closed-system boundary condition.

Returns A new *ClosedSystem* object

classmethod `fixed_compositions(unit_enum: tc_python.diffusion.Unit = <Unit.MASS_PERCENT: 3>)`

Returns a fixed-composition boundary condition.

Parameters `unit_enum` – The composition unit

Returns A new *FixedCompositions* object

classmethod `mixed_zero_flux_and_activity()`

Returns a mixed zero-flux and activity boundary condition, i.e. for the defined species different conditions are used.

Returns A new *MixedZeroFluxAndActivity* object

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.

classmethod `double_geometric(no_of_points: int = 50, lower_geometrical_factor: float = 1.1, upper_geometrical_factor: float = 0.9)`

Creates a double geomtric 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.

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)
Creates a 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)
Creates an 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.AbstractSolver`
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. 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

class `tc_python.diffusion.ClosedSystem`
Bases: `tc_python.diffusion.AbstractBoundaryCondition`
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

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

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. A detailed definition of the axis variables can be found in the 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

Returns A tuple of two lists of floats (time [s], mass fraction of the specified component)

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 (*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 (*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 hte phase at the specified time)

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_###`) will be 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

- **component** – The name of the component
- **phase** – The name of the phase

Returns A tuple of two lists of floats (time [s], total mole fraction of the component in the phase)

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.

Parameters **phase** – The name of the phase

Returns A tuple of two lists of floats (time [s], total mole fraction of the phase)

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.

Parameters **phase** – The name of the phase

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

get_values_of (*x_axis*: `Union[tc_python.quantity_factory.DiffusionQuantity, str]`, *y_axis*: `Union[tc_python.quantity_factory.DiffusionQuantity, str]`, *plot_condition*: `Union[tc_python.quantity_factory.PlotCondition, str]` = "", *independent_variable*: `Union[tc_python.quantity_factory.IndependentVariable, str]` = "") → `[typing.List[float], typing.List[float]]`

Returns the specified result from the simulation, allows all possible settings.

Note: As an alternative, DICTRA-console 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. Not every combination of settings will return a result.

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]

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.

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])

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.

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])

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 into a new TC-Python session 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

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 command for execution. These commands are executed after all other configuration directly before the calculation starts to run. All commands will be stored and used until explicitly deleted using `tc_python.diffusion.DiffusionIsoThermoCalculation.remove_all_console_commands()`.

Parameters `console_command` – The DICTRA console command

Returns This `DiffusionCalculation` 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.

Parameters `region` – The region to be added

Returns This `DiffusionCalculation` object

calculate () → `tc_python.diffusion.DiffusionCalculationResult`

Runs the diffusion calculation.

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

remove_all_console_commands ()

Removes all previously added console commands.

Returns This `DiffusionCalculation` object

set_simulation_time (*simulation_time: float*)

Sets the simulation time.

Parameters `simulation_time` – The simulation time [s]

Returns This `DiffusionCalculation` 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 modelling a tube geometry [m]

Returns This `DiffusionCalculation` object

with_left_boundary_condition (*boundary_condition: tc_python.diffusion.BoundaryCondition*)

Defines the boundary condition on the left edge of the system.

Default: A closed-system boundary condition

Parameters `boundary_condition` – The boundary condition

Returns This `DiffusionCalculation` object

with_options (*options: tc_python.diffusion.Options*)

Sets the general simulation conditions.

Parameters `options` – The general simulation conditions

Returns This `DiffusionCalculation` object

with_planar_geometry ()

Sets geometry to *planar*.

This is default.

Returns This `DiffusionCalculation` object

with_reference_state (*element: str, phase: str = 'SER', temperature: float = 'current_temperature', 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

with_right_boundary_condition (*boundary_condition: tc_python.diffusion.BoundaryCondition*)
Defines the boundary condition on the right edge of the system.

Default: A closed-system boundary condition

Parameters **boundary_condition** – The boundary condition

Returns This `DiffusionCalculation` object

with_solver (*solver: tc_python.diffusion.Solver*)

Sets the solver to use (*Classic, Homogenization* or *Automatic*). **Default is Automatic.**

Parameters **solver** – The solver to use

Returns This `DiffusionCalculation` 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 modelling a hollow sphere geometry [m]

Returns This `DiffusionCalculation` object

with_timestep_control (*timestep_control: tc_python.diffusion.TimestepControl*)

Sets the timestep control options.

Parameters **timestep_control** – The new timestep control options

Returns This `DiffusionCalculation` 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 command for execution. These commands are executed after all other configuration directly before the calculation starts to run. All commands will be stored and used until explicitly deleted using `tc_python.diffusion.DiffusionNonIsoThermalCalculation.remove_all_console_commands()`.

Parameters **console_command** – The DICTRA console command

Returns This `DiffusionCalculation` 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.

Parameters **region** – The region to be added

Returns This `DiffusionCalculation` object

calculate () → `tc_python.diffusion.DiffusionCalculationResult`

Runs the diffusion calculation.

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

remove_all_console_commands ()

Removes all previously added console commands.

Returns This `DiffusionCalculation` object

set_simulation_time (*simulation_time: float*)

Sets the simulation time.

Parameters **simulation_time** – The simulation time [s]

Returns This `DiffusionCalculation` 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 modelling a tube geometry [m]

Returns This `DiffusionCalculation` object

with_left_boundary_condition (*boundary_condition: tc_python.diffusion.BoundaryCondition*)

Defines the boundary condition on the left edge of the system.

Default: A closed-system boundary condition

Parameters **boundary_condition** – The boundary condition

Returns This `DiffusionCalculation` object

with_options (*options: tc_python.diffusion.Options*)

Sets the general simulation conditions.

Parameters `options` – The general simulation conditions

Returns This `DiffusionCalculation` object

with_planar_geometry ()

Sets geometry to *planar*.

This is default.

Returns This `DiffusionCalculation` object

with_reference_state (`element: str, phase: str = 'SER', temperature: float = 'current_temperature', 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`)

Defines the boundary condition on the right edge of the system.

Default: A closed-system boundary condition

Parameters `boundary_condition` – The boundary condition

Returns This `DiffusionCalculation` object

with_solver (`solver: tc_python.diffusion.Solver`)

Sets the solver to use (*Classic*, *Homogenization* or *Automatic*). **Default is Automatic.**

Parameters `solver` – The solver to use

Returns This `DiffusionCalculation` 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 modelling a hollow sphere geometry [m]

Returns This `DiffusionCalculation` 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*)

Sets the timestep control options.

Parameters **timestep_control** – The new timestep control options

Returns This `DiffusionCalculation` 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.AbstractCalculatedGrid`

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

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 `function` (*dictra_console_mode_function*: str)

Creates a initial concentration profile defined by a function in DICTRA-console syntax.

Parameters `dictra_console_mode_function` – The function, expressed in DICTRA-console mode syntax.

Returns A new *StepProfile* object

Note: This is an advanced feature, preferably a complex concentration profile should be generated using Python-libraries and added to the simulation using *tc_python.diffusion.PointByPointGrid*.

classmethod `linear` (*start*: float, *end*: float)

Creates a linear initial concentration profile.

Parameters

- **start** – Composition at the left side of the region [unit as defined in *CompositionProfile*].
- **end** – 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)

Creates an 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.FixedCompositions (unit_enum: tc_python.diffusion.Unit = <Unit.MASS_PERCENT: 3>)
```

Bases: *tc_python.diffusion.AbstractBoundaryCondition*

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]

```
class tc_python.diffusion.FunctionProfile (dictra_console_mode_function: str)
```

Bases: *tc_python.diffusion.AbstractElementProfile*

Creates an initial concentration profile defined by a function in DICTRA-console syntax.

Note: This is an advanced feature, preferably a complex concentration profile should be generated using Python-libraries and added to the simulation using *tc_python.diffusion.PointByPointGrid*.

get_type () → str
The type of the element profile.

Returns The type

```
class tc_python.diffusion.GeometricGrid (no_of_points: int = 50, geometrical_factor: float = 1.1)
```

Bases: *tc_python.diffusion.AbstractCalculatedGrid*

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.HomogenizationFunction`
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.HomogenizationSolver`

Bases: `tc_python.diffusion.AbstractSolver`

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.diffusionAutomaticSolver` instead if you do not need that behaviour.

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_homogenization_function (*homogenization_function_enum*:
`tc_python.diffusion.HomogenizationFunction = <HomogenizationFunction.RULE_OF_MIXTURES: 3>`)
Sets the *homogenization function* used by the *homogenization model*.

Default is `RULE_OF_MIXTURES`.

Parameters `homogenization_function_enum` – The homogenization function used by the homogenization model

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_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.LinearGrid` (*no_of_points*: *int = 50*)
Bases: `tc_python.diffusion.AbstractCalculatedGrid`

Represents an equally spaced grid.

get_no_of_points () → *int*
Returns the number of grid points.

Returns The number of grid points

get_type () → *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*: *float*, *end*: *float*)
Bases: `tc_python.diffusion.AbstractElementProfile`

Represents a linear initial concentration profile.

get_type () → *str*
The type of the element profile.

Returns The type

class `tc_python.diffusion.MixedZeroFluxAndActivity`

Bases: `tc_python.diffusion.AbstractBoundaryCondition`

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*: float)

Sets a fixed activity for an element at the boundary.

Parameters

- **element_name** – The name of the element
- **activity** – The activity

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. **This is the default setting.**

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_forced_starting_values_in_equilibrium_calculations ()

Enables forced start values for the equilibrium calculations. **The default is disabled forcing of start values.**

Returns This *Options* object

enable_save_results_to_file (*every_nth_step*: *Optional[int]* = None)

Enables and configures saving of results to file during the simulation. They can be saved for every n-th or optionally for every timestep (*None*). **Default:** Saving of the results at every timestep

Parameters **every_nth_step** – *None* or a value ranging from 0 to 99

Returns This *Options* object

enable_time_integration_method_euler_backwards ()

Enables *Euler backwards* integration. **The default method is trapezoidal integration.**

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. **This is the default method.**

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_interfaces (*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 than one phase.

Note: The first added phase represents the matrix phase, while all later added phases are *spheroid phases*, i.e. precipitate phases.

add_phase (*phase_name: str*)

Adds a phase to the region, each region must contain at least one phase.

Note: The first added phase represents the matrix phase, while all later added phases are *spheroid phases*, i.e. precipitate phases.

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

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.SimulationTime`

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()`

Returns an *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()`

Returns a *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()`

Returns a *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 behaviour.

Returns A new *HomogenizationSolver* object

class `tc_python.diffusion.StepProfile` (*lower_boundary: float, upper_boundary: float, step_at: float*)

Bases: *tc_python.diffusion.AbstractElementProfile*

Represents an initial constant concentration profile with a step at the specified position.

get_type() → 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. **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. **Not enabled by default.**

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

Note: For example, if 2 is entered the maximum time step is twice as long as the previous time step taken.

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`

Bases: `enum.Enum`

Represents a composition unit.

MASS_FRACTION = 2

Mass fraction.

MASS_PERCENT = 3

Mass percent.

MOLE_FRACTION = 0

Mole fraction.

MOLE_PERCENT = 1

Mole percent.

```
U_FRACTION = 4
U fraction
```

5.1.6 Module “propertymodel”

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()`.

add_poly_command (*poly_command: str*)

Registers a POLY console command for execution. These commands are executed after all other configuration directly before the calculation starts to run. All commands will be stored and used until explicitly deleted using `remove_all_poly_commands()`.

Parameters `poly_command` – The POLY console command

Returns This `PropertyModelCalculation` 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 POLY-commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten parenthesis, ...).

calculate () → `tc_python.propertymodel.PropertyModelResult`

Runs the property model calculation.

Returns A `PropertyModelResult` which later can be used to get specific values from the simulation.

get_argument_default (*argument_id*) → 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)

get_argument_description (*argument_id*) → 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

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 in the context of TC-Python.

Returns The ids of the available arguments

get_model_description () → str

Returns the description text of the current model.

Returns the description

get_model_parameter_value (*model_parameter_id*) → 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]

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

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 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

set_composition_unit (*unit_enum*: *tc_python.utils.CompositionUnit* = *<CompositionUnit.MOLE_PERCENT: 1>*)

Sets the composition unit.

Default: Mole percent (*CompositionUnit.MOLE_PERCENT*).

Parameters *unit_enum* – The new composition unit

Returns This *PropertyModelCalculation* object

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

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

set_model_parameter (*model_parameter_id*, *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

set_temperature (*temperature: float = 1000*)

Sets the temperature.

Default: 1000 K

Parameters `temperature` – The temperature [K]

Returns This *PropertyModelCalculation* object

class `tc_python.propertymodel.PropertyModelResult` (*result*)

Bases: *tc_python.abstract_base.AbstractResult*

The result of a property model calculation.

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

get_result_quantity_description (*result_quantity_id*) → 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

get_value_of (*result_quantity_id: 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]

save_to_disk (*path: str*)

Saves the result to disk. The result can later be loaded into a new TC-Python session 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

5.2 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 System-Builder is that the operations are performed on all the previously selected databases. The system is then used to create calculations.

deselect_phase (*phase_name_to_deselect: str*)

Deselects a phase for both the thermodynamic and the kinetic database.

Parameters `phase_name_to_deselect` – The phase name

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, they will be independent.

Returns A new *System* 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

without_default_phases ()

Removes all the default phases from both the thermodynamic and the kinetic database, any phase now needs to be selected manually for the databases.

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.

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_ges_parameter (*parameter: str*) → str

Returns a GES-database parameter expression from the database. Example: `system.get_ges_parameter("G(LIQUID,FE;0)")` might return the expression `"+1.2*GFELIQ"`.

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

Note: Please consult the Thermo-Calc GES-system documentation for details about the syntax.

Parameters **parameter** – The GES-parameter (for example: `"G(LIQUID,FE;0)"`)

Returns The GES-expression (for example: `"+1.2*GFELIQ"`)

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_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

run_ges_command (*command: str*)

Sends a command to the GES monitor. 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: The current *System* is copied and the GES-command is executed on the new system, i.e. the returned *System* object is independent from the current one.

Note: Please consult the Thermo-Calc GES-system documentation for details about the syntax.

Parameters **command** – The GES-command (for example: “AM-PH-DE FCC_A1 C_S 2 Fe:C”)

Returns A new *System* object

set_ges_parameter (*parameter: str, expression: str*)

Resets a GES-database parameter expression. Example: `system.set_ges_parameter("G(LIQUID,FE;0)", "+1.2*GFELIQ")`.

Note: The current *System* is copied and the GES-parameter is changed in the new system, i.e. the returned *System* object is independent from the current one.

Note: Please consult the Thermo-Calc GES-system documentation for details about the syntax.

Parameters **parameter** – The GES-parameter (for example: “G(LIQUID,FE;0)”)

Returns The GES-expression (for example: “+1.2*GFELIQ”)

Returns A new *System* object

with_cct_precipitation_calculation () → `tc_python.precipitation.PrecipitationCCTCalculation`
Creates a CCT-diagram calculation.

Returns A new `PrecipitationCCTCalculation` object

with_isothermal_diffusion_calculation () → `tc_python.diffusion.DiffusionIsoThermalCalculation`
Creates an isothermal diffusion calculation.

Returns A new `DiffusionIsoThermalCalculation` object

with_isothermal_precipitation_calculation () → `tc_python.precipitation.PrecipitationIsoThermalCalculation`
Creates an isothermal precipitation calculation.

Returns A new `PrecipitationIsoThermalCalculation` object

with_multiple_equilibria_calculation () → `tc_python.multi_equilibrium.MultipleEquilibriumCalculation`
Creates a “multi-equilibria” 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 higher performance for larger composition arrays.

Returns A new `MultipleEquilibriumCalculation` object

with_non_isothermal_diffusion_calculation () → `tc_python.diffusion.DiffusionNonIsoThermalCalculation`
Creates a non-isothermal precipitation calculation.

Returns A new `PrecipitationNonIsoThermalCalculation` object

with_non_isothermal_precipitation_calculation () → `tc_python.precipitation.PrecipitationNonIsoThermalCalculation`
Creates a non-isothermal precipitation calculation.

Returns A new `PrecipitationNonIsoThermalCalculation` object

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, ...]`), 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

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, ...]`), 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

with_property_model_calculation (*model*: *str*, *path_to_models*: *str* = *''*) → `tc_python.propertymodel.PropertyModelCalculation`
Creates a property model calculation.

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 model folder used by the normal Thermo-Calc application is used.

Returns A new `PropertyModelCalculation` object

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

```
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: $[AL_2O_3, \dots]$), 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 () → 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.

```
deselect_phase (phase_name_to_deselect: str)
Deselects a phase in the last specified database only.
```

Parameters **phase_name_to_deselect** – The name of the phase

Returns This *SystemBuilder* 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, they will be 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, they will be independent.
```

Returns A new *System* object

```
select_database_and_elements (database_name: str; list_of_element_strings: List[str])
Selects 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_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).

Parameters

- **path_to_user_database** – The path to the database file (*.TDB), defaults to the current working directory. Only the filename is required if the database is located in the same folder as the Python script.
- **list_of_element_strings** – A list of one or more elements as strings, for example ["Fe", "C"]

Returns This *SystemBuilder* object

without_default_phases ()

Deselects 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.Element` (*element*)

Bases: `object`

Represents an element, making detailed information about the element accessible.

get_name () → `str`

Returns the name of the element.

Returns The element name

is_interstitial () → `bool`

Returns if the element is interstitial.

Note: In the diffusion simulations (DICTRA), the assumption that the volume is carried by the substitutional elements only is applied. The interstitial elements are assumed to have zero molar volumes.

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_liquid () → `bool`

Returns if the phase is a liquid or ionic liquid phase.

Returns If the phase is a liquid phase

class `tc_python.entities.PhaseType`

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 an 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

is_interstitial () → bool

Returns if the species is interstitial.

Note: In the diffusion simulations (DICTRA), the assumption that the volume is carried by the substitutional elements only is applied. The interstitial elements are assumed to have zero molar volumes.

Returns If the species is interstitial

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.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 one TC-Python session, i.e. within one *with*-clause, caching will be anyway 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 will be listed.

Returns List of the available databases

get_property_models (*path_to_models: str = "*) → Set[str]

Lists the names of all property modes 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

load_result_from_disk ()

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 with TC-Python.

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.

Parameters

- **path_to_user_database** – The path to the database file (*.TDB), defaults to the current working directory. Only filename is required if the database is located in the same folder as the Python script.
- **list_of_elements** – The list of the selected elements in that database, for example ["Fe", "C"]

Returns A new `SystemBuilder` object

set_cache_folder (*path: str = "", precision_for_floats: int = 12*)

Sets a folder where results from calculations, and state of systems will be 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 python 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

set_ges_version (*version=<class 'int'>*)

set_log_level_to_debug ()

Sets log level to DEBUG

Returns This `SetUp` object

set_log_level_to_info ()

Sets log level to INFO

Returns This `SetUp` object

set_poly_version (*version=<class 'int'>*)

class `tc_python.server.TCPython` (*screen_log_handler=True, debug_logging=False, do_throw_on_backend_hard_crash=True, bug_mode=False*)

Bases: `object`

Starting point of the API. Typical syntax:

```
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. Due to the overhead that only makes sense if the calculation time per loop iteration is longer than a minute.

```
tc_python.server.start_api_server(screen_log_handler=True,                                de-
                                bug_mode=False,                                       is_unittest=False,
                                do_throw_on_backend_hard_crash=True)
```

Starts a process of the API server and sets up the socket communication with it.

Parameters

- **screen_log_handler** – If `True` adds a Python log-handler for printing all messages from the calculation to the screen. **Default:** `True`. Note that the handlers can also be adapted through the `tc_python.LOGGER` object at any time.
- **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..*

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.stop_api_server()
```

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 `TCPython`.**

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*)

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 will be used when this setting is set to False.

Returns A new `ActivityOfComponent` object.

classmethod chemical_diffusion_coefficient (*phase: str*, *diffusing_element: str*, *gradient_element: str*, *reference_element: str*)

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*)

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 will be used when this setting is set to False.

Param *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 `ChemicalPotentialOfComponent` object.

classmethod distance (*region: str = 'All'*)

Creates a quantity representing the distance [m].

Parameters *region* – The name of the region or *All* to choose global.

classmethod intrinsic_diffusion_coefficient (*phase: str*, *diffusing_element: str*, *gradient_element: str*, *reference_element: str*)

Creates a quantity representing the intrinsic 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 `IntrinsicDiffusionCoefficient` object.

classmethod `l_bis` (*phase: str, diffusing_element: str, gradient_element: str, reference_element: str*)

Creates a quantity representing L' 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 = '*'*)

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 = '*'*)

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 `mobility_of_component_in_phase` (*phase: str, component: str*)

Creates a quantity representing the mobility of a component in a phase [m^2/Js].

Parameters

- **phase** – The name of the phase
- **component** – The name of the component

Returns A new `MobilityOfComponentInPhase` object.

classmethod `mole_fraction_of_a_component` (*component: str = '*'*)

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 = '*'*)

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 `position_of_lower_boundary_of_region` (*region: str*)

Creates a quantity representing the position of lower boundary of a region [m].

Parameters **region** – The name of the region

Returns A new `PositionOfLowerBoundaryOfRegion` object.

classmethod position_of_upper_boundary_of_region (*region: str*)

Creates a quantity representing the position of upper boundary of a region [m].

Parameters **region** – The name of the region

Returns A new `PositionOfUpperBoundaryOfRegion` object.

classmethod temperature ()

Creates a quantity representing the temperature [K].

Returns A new `Temperature` object.

classmethod thermodynamic_factor (*phase: str, diffusing_element: str, gradient_element: str, reference_element: str*)

Creates a quantity representing thermodynamic factor of a phase.

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 `ThermoDynamicFactor` object.

classmethod time ()

Creates a quantity representing the time [s].

classmethod total_mass_fraction_of_component (*component: str*)

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*)

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*)

Creates a quantity representing the total mass fraction of a phase.

Param **phase**: The name of the phase.

Returns A new `TotalMassFractionOfPhase` object.

classmethod total_mole_fraction_of_component (*component: str*)

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*)

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*)

Creates a quantity representing the total volume fraction of a phase.

Param `phase`: The name of the phase.

Returns A new `TotalVolumeFractionOfPhase` object.

classmethod `tracer_diffusion_coefficient` (*phase: str, diffusing_element: str*)

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.

classmethod `u_fraction_of_a_component` (*component: str*)

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*)

Creates a quantity representing a user defined function.

Parameters `expression` – The function expression

Returns A new `Function` object

classmethod `velocity_of_lower_boundary_of_region` (*region: str*)

Creates a quantity representing the velocity of lower boundary of a region [m/s].

Parameters `region` – The name of the region

Returns A new `VelocityOfLowerBoundaryOfRegion` object.

classmethod `velocity_of_upper_boundary_of_region` (*region: str*)

Creates a quantity representing the velocity of upper boundary of a region [m/s].

Parameters `region` – The name of the region

Returns A new `VelocityOfUpperBoundaryOfRegion` object.

classmethod `width_of_region` (*region: str*)

Creates a quantity representing the width of a region [m].

Parameters `region` – The name of the region

Returns A new `WidthOfRegion` object.

class `tc_python.quantity_factory.IndependentVariable`

Bases: `tc_python.quantity.AbstractQuantity`

Factory class providing quantities used for defining the independent variable in general diffusion result querying.

classmethod `distance` (*region: str = 'All'*)

Creates an independent variable representing the distance [m].

Returns A new *Distance* object

classmethod time ()

Creates an independent variable representing the time [s].

Returns A new *Time* object

class `tc_python.quantity_factory.PlotCondition`

Bases: `tc_python.quantity.AbstractQuantity`

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 (*region: str = 'All'*)

Creates a plot condition representing the distance [m].

Parameters **region** – The name of the region or *All* to choose global.

Returns A new *DistanceCondition* object

classmethod integral ()

Creates an integral plot condition.

Returns A new *IntegralCondition* object

classmethod interface (*region: str, interface_position: tc_python.utils.InterfacePosition*)

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 (*timepoints: Union[List[Union[float, str]], float, str] = 'Last'*)

Creates a plot condition representing the time [s].

Parameters **timepoints** – The timepoints, can be a single value or a list of values. 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 ()

Creates a quantity representing the apparent heat capacity [J/g/K].

Returns A new *ApparentHeatCapacityPerGram* object.

classmethod apparent_heat_capacity_per_mole ()

Creates a quantity representing the apparent heat capacity [J/mol/K].

Returns A new *ApparentHeatCapacityPerMole* object.

classmethod `apparent_volumetric_thermal_expansion_coefficient()`

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 = '*'*)

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 = '*'*)

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_solid_phase` (*phase: str = '*'*)

Creates a quantity representing the average density of a solid phase [g/cm³].

Parameters **phase** – The name of the phase or `ALL_PHASES` to choose all solid phases

Returns A new `DensityOfSolidPhase` object.

classmethod `density_of_system()`

Creates a quantity representing the average density of the system [g/cm³].

Returns A new `DensityOfSystem` object.

classmethod `distribution_of_component_of_phase` (*phase: str*, *component: str*)

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()`

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()`

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()`

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()`

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 = '*')`

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

Returns A new `MassFractionOfASolidPhase` object.

classmethod `mass_fraction_of_all_liquid()`

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()`

Creates a quantity representing the total mass fraction of all solid phases.

Returns A new `MassFractionOfAllSolidPhase` object.

classmethod molar_volume_of_phase (*phase: str = '*'*)

Creates a quantity representing the molar volume of a phase [m³/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 ()

Creates a quantity representing the molar volume of the system [m³/mol].

Returns A new `MolarVolumeOfSystem` object.

classmethod mole_fraction_of_a_solid_phase (*phase: str = '*'*)

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 ()

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 ()

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 = None*)

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 do start with 1.**

Returns A new `SiteFractionOfComponentInPhase` object.

classmethod temperature ()

Creates a quantity representing the temperature [K].

Returns A new `Temperature` object.

classmethod user_defined_function (*expression: str*)

Creates a quantity representing a user defined function.

Parameters **expression** – The function expression

Returns A new `Function` 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*)

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 will be used when this setting is set to False.

Param `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 `ActivityOfComponent` object.

classmethod `chemical_diffusion_coefficient` (*phase: str*, *diffusing_element: str*, *gradient_element: str*, *reference_element: str*)

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*)

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 will be used when this setting is set to False.

Param `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 `ChemicalPotentialOfComponent` object.

classmethod `composition_of_phase_as_mole_fraction` (*phase: str = '*'*, *component: str = 'All'*)

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 = '*'*)

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*)

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 = '*'*)

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 = '*'*)

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 = '*'*)

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 = '*'*)

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 = '*'*)

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()`

Creates a quantity representing the pressure [Pa].

Returns A new `Pressure` object.

classmethod `system_size()`

Creates a quantity representing the system size [mol].

Returns A new `SystemSize` object.

classmethod `temperature()`

Creates a quantity representing the temperature [K].

Returns A new `Temperature` object.

classmethod `tracer_diffusion_coefficient(phase: str, diffusing_element: str)`

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.

classmethod `u_fraction_of_a_component(component: str)`

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)`

Creates a quantity representing a user defined function.

Parameters `expression` – The function expression

Returns A new `Function` object

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”

class `tc_python.utils.CompositionUnit`

Bases: `enum.Enum`

The composition unit.

MASS_FRACTION = 2

Mass fraction.

MASS_PERCENT = 3

Mass percent.

MOLE_FRACTION = 0

Mole fraction.

MOLE_PERCENT = 1

Mole percent.

class `tc_python.utils.Condition` (*console_mode_syntax: str, value: float*)

Bases: `object`

A condition expressed in Console mode syntax (e.g. “X(Cr)”).

class `tc_python.utils.InterfacePosition`

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.ResultValueGroup` (*result_line_group_java*)

Bases: `object`

A x-y-dataset representing a line data calculation result (i.e. a Thermo-Calc *quantity 1* vs. *quantity 2*).

Warning: Depending on the calculator, the dataset might contain *NaN*-values to separate the data between different subsets.

Variables

- **x** – list of floats representing the first quantity (“x-axis”)
- **y** – list of floats representing the second quantity (“y-axis”)

class `tc_python.utils.TemperatureProfile`

Bases: `object`

Represents a time-temperature profile used by non-isothermal calculations.

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.

add_time_temperature (*time: float, temperature: float*)

Adds a time-temperature point to the non-isothermal temperature profile.

Parameters

- **time** – The time [s]
- **temperature** – The temperature [K]

Returns This `TemperatureProfile` object

5.7 Module “exceptions”

exception `tc_python.exceptions.APIServerErrorException`

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.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.raise_python_exceptions` (*func*)

Internal method of the API: Usage of that decorator maps all relevant Java exceptions in the API to the appropriate Python exception.

5.8 Module “abstract_base”

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.

<p>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.</p>

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.

TROUBLESHOOTING

This section provides an FAQ for common problems that occur when using TC-Python.

6.1 “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 *TC-Python Quick Install 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

```
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*):

```
Microsoft Windows [Version 10.0.16299.431]
(c) 2017 Microsoft Corporation. All rights reserved.

(venv) C:\Users\User\Documents\>
```

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 *TC-Python Quick Install 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*.

6.2 “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.

1. TC-Python has a dependency to the package *py4j*. *pip* will resolve that by downloading it 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
```

2. Another alternative is to download the latest *.whl-file of the *py4j* package from the repository server 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 that updates to that package have to be fully manual also in the future.

PYTHON MODULE INDEX

t

`tc_python.abstract_base`, 125
`tc_python.diffusion`, 69
`tc_python.entities`, 104
`tc_python.exceptions`, 124
`tc_python.precipitation`, 28
`tc_python.propertymodel`, 95
`tc_python.quantity_factory`, 111
`tc_python.scheil`, 45
`tc_python.server`, 107
`tc_python.single_equilibrium`, 21
`tc_python.step_or_map_diagrams`, 50
`tc_python.system`, 99
`tc_python.utils`, 122