TC-Python Documentation

Release 2018a

Thermo-Calc Software AB

Feb 28, 2018

CONTENTS

CHAPTER

ONE

TC-PYTHON QUICK INSTALL GUIDE

This quick guide helps you do a TC-Python API installation.

There is a PDF guide included with your installation. In Thermo-Calc from the 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, especially if you have other Python installations. On Linux and Mac use the system Terminal and a virtual environment, see the python glossary: https://docs.python.org/3/glossary.html#term-virtual-environment.

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\2018a\SDK\TC-Python\`

   Details for Mac and Linux installations are described in Default Directory Locations in the *Thermo-Calc Installation Guide*.
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 the IntelliSense, 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 and set up the pip install path, 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\2018a\SDK\TC-Python\`

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

2. Click 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** .

CHAPTER

TWO

MAC OS: SETTING ENVIRONMENT VARIABLES

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

```
TC18A_HOME=/Applications/Thermo-Calc-2018a.app/Contents/Resources
```

If you use a license server:

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

If you have a node-locked license:

```
LSHOST= NO-NET   LSERVRC=/Applications/Thermo-Calc-2018a.app/Contents/
Resources/lservrc
```

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

Select **Run→Edit Configurations** to open the **Run/Debug Configurations** window. Enter the environment variable(s) by clicking the button to the right of the **Environment Variables** text field.

CHAPTER

THREE

HIGH LEVEL ARCHITECTURE

TC-Python contains classes of these types:

- **TCPython** – which is where you start
- **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: This starts a process running a calculation server. Your code will then, via TC-Python, use socket communication to send and receive messages to and from that server. This is for information only.

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

```
# after with clause
```

the calculation server automatically shuts down, and all temporary files will be deleted. To ensure that this happens, it is important that you structure your Python code using a `with()` clause, as the example above shows.

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 with which 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 can not 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()` on that.

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 Precipitation calculations

Everything that you can configure in the *Precipitation Calculator* in Thermo-Calc Graphical Mode, you can configure in this calculation. For the calculation to be possible, you need at least to enter a matrix phase, a precipitate phase, temperature, simulation time and compositions.

Example:

```
from tc_python import *

with TCPython() as start:
      r = (
            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)
```

(continues on next page)

(continued from previous page)

```
                    .set_simulation_time(1e5)
                    .with_matrix_phase(MatrixPhase("FCC_A1")
                                       .add_precipitate_phase(PrecipitatePhase(
↪"AL3SC"))
                                       )
                .calculate()
               )
```

3.3.2 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 will no longer give values corresponding to the first calculation.

This is different from how other calculations work. If you want a **Result** that you can use later, you need to call `calculate_with_state()`.

Note: `calculate()` has MUCH better performance than `calculate_with_state()`, and works for almost all situations.

Example

```
from tc_python import *

with TCPython() as session:
    g = (
        session.
            select_database_and_elements("FEDEMO", ["Fe", "Cr", "C"]).
            get_system().
            with_single_equilibrium_calculation().
                set_condition("P", 100000.0).
                set_condition("T", 2000.0).
                set_condition("N", 1.0).
                set_condition("X(Cr)", 0.1).
                set_condition("X(C)", 0.01).
            calculate().
            get_value_of("G")
        )
```

3.4 Result

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

The **Result** class returned has different methods, depending on the type of calculation.

The **Result** is used to get numerical values from a calculation that has run.

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.

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()

# Now its possible to get results from the single equilibrium calculation,
# without having to re-run it

gibbs = single_eq_result.get_value_of("G")
```

In other words. You can mix different calculations and results without having to think about which state the calculation server is in.

CHAPTER

FOUR

API REFERENCE

4.1 Module system

class tc_python.system.**MultiDatabaseSystemBuilder**(*multi_database_system_builder*)
Bases: object

Used to select databases, elements, phases etc. and create a System object. The difference to the class SystemBuilder is that the operations are performed on all the previously selected databases. The system is then used to create calculations.

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, the 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 is has been created. If you want to change the system, you must instead create a new one.

get_all_phases_in_databases() → List[str]
Returns all phases present in the selected databases, regardless on selected elements, phases etc.

Returns a list of phases

get_phases_in_system() → List[str]
Returns all phases present in the system due to its configuration (selected elements, phases, etc.).

Returns a list of phases

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

Returns A new `PrecipitationCCTCalculation` object

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

Returns A new `PrecipitationIsoThermalCalculation` object

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

Returns A new `PrecipitationNonIsoThermalCalculation` object

with_single_equilibrium_calculation() → tc_python.single_equilibrium.SingleEquilibriumCalculation
Creates a single equilibrium calculation.

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

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

4.2 Module server

class tc_python.server.**SetUp**
Bases: object

Starting point for all calculations.

Note: This class exposes methods that have no precondition, it is used for choosing databases and elements.

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.MultiDatabaseSystemBui

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

class tc_python.server.**TCPython**(*debug_mode=False*)

Bases: `object`

Starting point of the API. Typical syntax:

```
with TCPython() as session:
    session.select_database_and_elements(...)
```

tc_python.server.**start_api_server**(*debug_mode=False*, *is_unittest=False*)

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

Parameters

- **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**.

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*.

4.3 Module single_equilibrium

class tc_python.single_equilibrium.**SingleEquilibriumCalculation**(*calculator*)
Bases: object

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.

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

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

remove_all_conditions()
Removes all set conditions.

Returns This *SingleEquilibriumCalculation* object

remove_condition(*console_mode_syntax: str*)
Removes the specified condition.

Parameters **console_mode_syntax** – The condition expressed in Console Mode syntax (e.g. "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.

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 *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(*console_mode_syntax: str*, *value: float*)
Sets the specified condition.

Parameters

- **console_mode_syntax** – The condition expressed in Console Mode syntax (e.g. "X(CR)").
- **value** – The value of the condition

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*)
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

class tc_python.single_equilibrium.**SingleEquilibriumResult**(*result*)
Bases: object

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

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

Returns The selected components

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 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 stable phases

get_value_of(*console_mode_expression: str*) → float
Returns a value from a single equilibrium calculation.

Parameters **console_mode_expression** – The required property expressed in Console Mode syntax (e.g. "NPM(FCC_A1)")

Returns The requested value

invalidate()
Invalidates the object and frees the disk space used by it.

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

---

class tc_python.single_equilibrium.**SingleEquilibriumTempResult**(*result*)
Bases: object

Result of a single equilibrium calculation that is only valid until something gets changed in the calculation state. It can be evaluated using 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.

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

Returns The selected components

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 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 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(*console_mode_expression: str*) → float
Returns a value from a single equilibrium calculation.

Parameters **console_mode_expression** – The required property expressed in Console Mode syntax (e.g. "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

invalidate()
Invalidates the object and frees the disk space used by it.

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

4.4 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.

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** –

set_dislocation_density(*dislocation_density: float*)
Enter a numerical value. The default is 5.0E12 m-2.

Parameters **dislocation_density** –

set_grain_aspect_ratio(*grain_aspect_ratio: float*)
Enter a numerical value. The default is 1.0.

Parameters **grain_aspect_ratio** –

set_grain_radius(*grain_radius: float*)
Sets grain size. The default is 1.0E-4 m

Parameters **grain_radius** –

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

set_mobility_enhancement_prefactor(*mobility_enhancement_prefactor: float*)
A parameter that multiplies to the mobility data from database. The default is 1.0

Parameters **mobility_enhancement_prefactor** –

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

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

with_elastic_properties_cubic(*c11: float*, *c12: float*, *c44: float*)
Sets elastic properties to cubic. **Default**: if not chosen, the default is DISREGARD :param c11: :param c12: :param c44:

with_elastic_properties_disregard()
Set to disregard to ignore the elastic properties. This is the default option. **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 :param shear_modulus: :param poisson_ratio:

class tc_python.precipitation.**NumericalParameters**
Bases: object

Numerical parameters

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

Parameters **max_overall_volume_change** –

set_max_radius_points_per_magnitude(*max_radius_points_per_magnitude: float*)
Sets max no. of grid points over one order of magnitude in radius. the default is 200.0

Parameters **max_radius_points_per_magnitude** –

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

Parameters **max_rel_change_critical_radius** –

set_max_rel_change_nucleation_rate_log(*max_rel_change_nucleation_rate_log: float*)
This parameter ensures accuracy for the evolution of effective nucleation rate. The default is 0.5.

Parameters **max_rel_change_nucleation_rate_log** –

set_max_rel_radius_change(*max_rel_radius_change: float*)
The maximum value allowed for relative radius change in one time step. The default is 0.01.

Parameters **max_rel_radius_change** –

set_max_rel_solute_composition_change(*max_rel_solute_composition_change: float*)
Set a limit on the time step by controlling solute depletion or saturation, especially at isothermal stage. The default is 0.01.

Parameters **max_rel_solute_composition_change** –

set_max_time_step(*max_time_step: float*)
The maximum time step allowed for time integration as fraction of the simulation time. The default is 0.1.

Parameters **max_time_step** –

set_max_time_step_during_heating(*max_time_step_during_heating: float*)
The upper limit of the time step that has been enforced in the heating stages. The default is 1.0 s.

Parameters **max_time_step_during_heating** –

set_max_volume_fraction_dissolve_time_step(*max_volume_fraction_dissolve_time_step: float*)
Sets max volume fraction of subcritical particles allowed to dissolve in one time step. The default is 0.01

Parameters **max_volume_fraction_dissolve_time_step** –

set_min_radius_nucleus_as_particle(*min_radius_nucleus_as_particle: float*)
The cut-off lower limit of precipitate radius. The default is 5.0E-10 m.

Parameters **min_radius_nucleus_as_particle** –

set_min_radius_points_per_magnitude(*min_radius_points_per_magnitude: float*)
Sets min no. of grid points over one order of magnitude in radius. The default is 100.0

Parameters **min_radius_points_per_magnitude** –

set_radius_points_per_magnitude(*radius_points_per_magnitude: float*)
Sets no. of grid points over one order of magnitude in radius. The default is 150.0

Parameters **radius_points_per_magnitude** –

set_rel_radius_change_class_collision(*rel_radius_change_class_collision: float*)
Sets relative radius change for avoiding class collision. The default is 0.5

Parameters **rel_radius_change_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: str*, *composition_value: float*)
Sets the initial precipitate composition.

Parameters

- **element** – 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/m^2]

Returns This *PrecipitatePhase* object

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

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

Parameters **interfacial_energy_estimation_prefactor** – The prefactor for the calculated interfacial energy

Returns This *PrecipitatePhase* object

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

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

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

Returns This *PrecipitatePhase* object

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

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

Returns This *PrecipitatePhase* object

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

Parameters

- **wetting_angle** – If not set, a default value of 90 degrees is used [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 m^4/(Js).

Parameters **phase_boundary_mobility** – The phase boundary mobility [m^4/(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.TransformationStrainCalculationOpti*)
Sets the transformation strain calculation option. **Default**: *TransformationStrainCalculationOption.DISREGARD*.

Parameters **transformation_strain_calculation_option_enum** – The chosen option

Returns This *PrecipitatePhase* object

set_wetting_angle(*wetting_angle: float*)
Sets the wetting angle. Only relevant if the activated nucleation sites use that setting. **Default**: A wetting angle of 90 degrees.

Parameters **wetting_angle** – The wetting angle [degrees]

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.precipitation.PrecipitationCalculation*

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_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

class tc_python.precipitation.**PrecipitationCalculation**(*calculation*)
Bases: object

Abstract base class for all precipitation calculations. Cannot be instantiated, use one of its subclasses instead.

set_composition(*element_name: str*, *value: float*)
Sets the composition of the elements. The default composition unit is mole percent, the unit for the composition can be changed using *set_composition_unit()*.

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, the default unit is mole percent (CompositionUnit.MOLE_PERCENT).

Parameters **unit_enum** – The new composition unit

Returns This *PrecipitationCalculation* 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: object

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

invalidate()
Invalidates the object and frees the disk space used by it.

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

---

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_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_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.precipitation.PrecipitationCalculation*

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_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

class tc_python.precipitation.**PrecipitationNonIsoThermalCalculation**(*calculation*)
Bases: *tc_python.precipitation.PrecipitationCalculation*

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_simulation_time(*simulation_time: float*)
Sets the simulation time.

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

Returns This PrecipitationNonThermalCalculation object

with_temperature_profile(*temperature_profile: tc_python.utils.TemperatureProfile*)
Sets the temperature profile to use with this calculation.

Parameters **temperature_profile** – the temperature profile object (specifying time / temperature points)

Returns This PrecipitationNonThermalCalculation object

class tc_python.precipitation.**PrecipitationTTTCalculation**(*calculation*)
Bases: *tc_python.precipitation.PrecipitationCalculation*

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_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

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%).

4.5 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.**TemperatureProfile**
Bases: object

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

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

4.6 Module exceptions

exception tc_python.exceptions.**APIServerException**
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.**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.**GeneralException**
Bases: *tc_python.exceptions.TCException*

A general exception that might occur in different situations.

exception tc_python.exceptions.**InvalidResultStateException**
Bases: *tc_python.exceptions.CalculationException*

Trying to access a `SingleEquilibriumTempResult` object that got already invalidated by condition changes.

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.**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.

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.

PYTHON MODULE INDEX

t