PyDelphin API-Referenz

Table of Contents

1. Zweck und Geltungsbereich . . . **1**
2. Projektdateien . . . **2**
2.1. Lesen und Schreiben . . . 2
2.2. `DelphinProject` . . . 2
2.3. Diskretisierung . . . 3
3. Materialien . . . **3**
3.1. Materialdateien . . . 3
3.2. `MaterialFile` . . . 3
3.3. Materialreferenzen im Projekt . . . 4
4. Diskretisierung und Schichtdicken . . . **4**
5. Solverausführung . . . **5**
5.1. `SolverRunConfig` . . . 5
5.2. `run_solver` . . . 5
5.3. `SolverRunResult` . . . 6
6. Variantenstudien . . . **6**
6.1. `MaterialReplacementStudy` . . . 6
6.2. `VariantStudy` . . . 6
6.3. Gemeinsame Studienergebnisse . . . 7
7. Standardisierte Evaluatoren . . . **7**
8. Ergebnis- und Logauswertung . . . **8**
9. Assessments . . . **8**
10. JSON-Konfiguration . . . **9**
11. Stabilität und Kompatibilität . . . **9**

1. Zweck und Geltungsbereich

Dieses Dokument beschreibt die öffentliche Python-Schnittstelle von PyDelphin. Es ist als Nachschlagewerk für eigene Scripts und Variantenstudien gedacht. Eine Einführung mit durchgängigen Beispielen steht im separaten **PyDelphin Scripting Tutorial**.

Die beschriebenen Namen werden aus `pydelphin` oder den ausdrücklich genannten Untermodulen importiert. Namen mit führendem Unterstrich gehören nicht zur öffentlichen Schnittstelle.

```
PYTHONPATH=/home/fechner/Delphin6/scripts/PyDelphin/src python3 mein_script.py
```

2. Projektdateien

2.1. Lesen und Schreiben

`read_project(path: str | Path) → DelphinProject`

Liest eine `.d6p`-Datei und gibt ein veränderbares `DelphinProject`-Objekt zurück. `path` muss auf eine vorhandene Projektdatei zeigen.

`parse_project_xml(xml_text: str, source_path: str | Path | None = None) → DelphinProject`

Liest eine Projektbeschreibung aus einem XML-String. `source_path` ist optional und dient zur Dokumentation der Quelle.

`project_to_xml(project: DelphinProject, *, include_declaration: bool = True) → str`

Serialisiert ein Projekt als XML-String.

`write_project(project: DelphinProject, path: str | Path, *, encoding: str = "utf-8") → None`

Schreibt ein Projekt in eine `.d6p`-Datei.

Beispiel:

```
from pydelphin import read_project, write_project

project = read_project("input/project.d6p")
project.project_info.values["Comment"] = "Variante A"
write_project(project, "output/project.d6p")
```

2.2. `DelphinProject`

`DelphinProject` ist das zentrale Projektmodell. Wichtige Attribute sind:

`root: xml.etree.ElementTree.Element`

XML-Wurzelelement für direkte, nicht durch Adapter abgedeckte Änderungen.

`source_path: Path | None`

Ursprünglicher Dateipfad, sofern bekannt.

`project_info: ProjectInfo`

Allgemeine Projektinformationen.

`discretization: Discretization`

Diskretisierung mit den Achsen und Zellweiten.

`material_references: MaterialsSection | None`

Im Projekt registrierte Materialien.

`assignments: AssignmentsSection | None`

Material-, Rand-, Anfangs- und Ausgabezuweisungen.

`conditions: ConditionsSection | None`

Definitionen von Klima-, Rand-, Feld-, Kontakt- und Interfacebedingungen.

`init: InitSection | None`

Simulations- und Solverparameter.

`section(name: str) → Element | None`

Liefert einen XML-Abschnitt nach Namen. Diese Methode ist für Erweiterungen gedacht, die noch keinen spezialisierten Adapter besitzen.

`placeholder_values() → dict[str, str]`

Liefert die im Projekt definierten Verzeichnis-Platzhalter.

`resolve_placeholders(value: str) → str`

Ersetzt bekannte Platzhalter in einem Pfad oder Text.

2.3. Diskretisierung

`Discretization.axis(name: str) → AxisSteps | None`

Liefert die Achse `X`, `Y` oder `Z`, sofern sie vorhanden ist.

`Discretization.dimension → int`

Gibt die räumliche Dimension des Projekts zurück.

`AxisSteps` enthält `axis`, `unit` und `values`. `values` ist eine Liste der Zellweiten in der jeweiligen Achse.

3. Materialien

3.1. Materialdateien

`read_material(path: str | Path, encoding: str = "utf-8") → MaterialFile`

Liest eine `.m6`-Materialdatei.

`parse_material(text: str, source_path: str | Path | None = None) → MaterialFile`

Liest eine Materialdatei aus Text.

`write_material(material: MaterialFile, path: str | Path | None = None, encoding: str | None = None) → None`

Schreibt ein Material. Wird `path` nicht angegeben, muss das Material einen passenden Zielpfad kennen.

3.2. `MaterialFile`

`section(name: str) → MaterialSection | None` und `require_section(name: str) → MaterialSection`:: Suchen einen Materialabschnitt; die `require`-Variante löst bei fehlendem Abschnitt einen Fehler aus.

`parameter(section_name: str, parameter_name: str) → MaterialParameter | None`

Liefert einen skalaren Materialparameter.

`spline(section_name: str, function_name: str) → MaterialSpline | None`

Liefert eine Materialfunktion.

`density()`, `heat_capacity()`, `theta_eff()`, `theta_por()`, `theta_80()`, `lambda_dry()`, `thermal_conductivity()`, `mew()`, `aw()`, `kleff()`, `kl_eff()` und `dleff()`:: Komfortzugriff auf häufig verwendete Materialparameter.

`set_density(value: float, unit: str = "kg/m3") → MaterialParameter`

`set_heat_capacity(value: float, unit: str = "J/kgK") → MaterialParameter`

Ändern grundlegender Materialparameter.

`set_storage_parameter(name: str, value: float, unit: str | None = None) → MaterialParameter`

`set_lambda(value: float, unit: str = "W/mK", direction: str = "U") → MaterialParameter`

Ändern von Speicher- und Wärmeleitparametern.

3.3. Materialreferenzen im Projekt

`project.material_references` ist eine `MaterialsSection`.

`by_name(name: str) → MaterialReference | None`

Sucht eine Referenz.

`require_name(name: str) → MaterialReference`

Sucht eine Referenz und meldet einen Fehler, wenn sie fehlt.

`by_path(path: str) → list[MaterialReference]`

Liefert alle Referenzen mit dem angegebenen Pfad.

`add_reference(name: str, path: str, color: str = "", hatch_code: str = "") → MaterialReference`

Fügt eine neue Materialreferenz hinzu.

`replace_assignments(old_name: str, new_name: str) → int`

Ersetzt alle Materialzuweisungen von `old_name`. Die neue Referenz muss existieren. Eine Selbstzuweisung wird ignoriert und liefert `0`.

`swap_assignments(first_name: str, second_name: str) → int`

Tauscht zwei Materialzuweisungen.

`set_name(old_name: str, new_name: str, update_assignments: bool = True) → MaterialReference`

`set_path(name: str, path: str) → MaterialReference`

`set_color(name: str, color: str | None) → MaterialReference`

`set_hatch_code(name: str, hatch_code: str | None) → MaterialReference`

Ändern die Eigenschaften einer vorhandenen Referenz.

4. Diskretisierung und Schichtdicken

`VariableDiscretizationConfig`

Konfiguration eines variablen 1D-Gitters.

`min_width: float = 0.005`

minimale Zellweite in Metern.

`max_width: float = 0.05`

maximale Zellweite in Metern.

`stretch: float = 1.25`

Gradierung der Zellen an den Schichtgrenzen.

`validate() → None` prüft die Werte.

`layers_1d(project: DelphinProject) → list[Layer1D]`

Rekonstruiert die zusammenhängenden Materialschichten eines 1D-Projekts.

`rediscretize_1d(project, config: VariableDiscretizationConfig | None = None, layer_thicknesses: list[float] | None = None) → list[Layer1D]`

Erzeugt ein variables X-Gitter und aktualisiert die betroffenen Material-, Rand- und Elementzuweisungen. Das Projekt wird direkt verändert. `layer_thicknesses` folgt der Reihenfolge aus `layers_1d()`.

5. Solverausführung

5.1. SolverRunConfig

`SolverRunConfig` steuert einen einzelnen Solverlauf:

`wall_time_limit_s: float = 1800.0`

maximale Echtzeit.

`progress_timeout_s: float = 300.0`

Abbruch, wenn sich die Simulationszeit so lange nicht weiterbewegt.

`poll_interval_s: float = 1.0`

Abstand zwischen zwei Logprüfungen.

`processes: int = 2`

an den Solver übergebene Prozessanzahl.

`termination_grace_s: float = 10.0`

Zeit zum sauberen Beenden vor dem erzwungenen Abbruch.

5.2. run_solver

`run_solver(project_path: str | Path, solver: str | Path, config: SolverRunConfig | None = None) → SolverRunResult`

Startet den Solver überwacht und wertet `progress.tsv` sowie `screenlog.txt` aus. Der Solver wird mit `-x`, `--verbosity-level=0` und `-p=<processes>` gestartet.

`SolverStatus` kann folgende Werte annehmen:

- `success`
- `solver_error`
- `wall_time_limit`

- `simulation_progress_timeout`
- `missing_progress`

5.3. `SolverRunResult`

Wichtige Ergebnisattribute sind:

`project_path`, `status`, `return_code`, `wall_time_s`, `simulation_time_s`, `progress_percent`, `progress_source`, `termination_reason`, `last_known_simulation_time_s`, `performance_estimate_s_per_s` und `has_complete_results`.

Auch ein abgebrochener Lauf liefert ein verwertbares Ergebnis. Dadurch können langsame oder nicht konvergierende Varianten miteinander verglichen werden.

6. Variantenstudien

6.1. `MaterialReplacementStudy`

`MaterialReplacementStudy(base_project, replacement_name, replacement_path, output_dir, *, replacement_color="", replacement_hatch_code="", reuse_existing=True)`

Erzeugt eine Baseline und je eine Variante für jedes verwendete Material, das nicht bereits dem Ersatzmaterial entspricht.

`variants() → list[MaterialVariant]`

Ermittelt die ersetzbaren Materialien, ohne Dateien zu erzeugen.

`generate() → list[tuple[MaterialVariant, Path]]`

Erzeugt die Variantenprojekte.

`run(solver, config=None, *, verbose=False, max_workers=1) → StudyResult`

Erzeugt und berechnet die Varianten.

6.2. `VariantStudy`

`VariantStudy(base_project, output_dir, *, reuse_existing=True)`

Grundlage für kombinierte Variantenstudien.

`add_parameter(name: str, options: Iterable[VariantOption | tuple[str, Any]]) → None`

Fügt einen Parameter mit benannten Optionen hinzu.

`variants() → list[VariantDefinition]`

Liefert das kartesische Produkt aller Parameteroptionen.

`generate(configure: ProjectConfigurator | None = None) → list[VariantDefinition]`

Erzeugt die Projekte. `configure(project, values, variant_directory)` kann projektabhängige Änderungen und das Kopieren von Zusatzdateien durchführen.

`run(solver, config=None, *, configure=None, verbose=False, max_workers=1) → StudyResult`

Erzeugt und berechnet die Varianten.

6.3. Gemeinsame Studienergebnisse

`ParallelSolverRunner(max_workers: int = 1)`

Führt Varianten mit begrenzter Parallelität aus.

`run(variants, solver, config=None, *, verbose=False) → StudyResult`

Berechnet alle erzeugten `VariantDefinition`-Objekte. Bei `verbose=True` werden Start und Abschluss jeder Variante ausgegeben.

`StudyResult`

Sequenz von `VariantRun`-Objekten.

`successful_runs → tuple[VariantRun, …]`

liefert nur erfolgreiche Läufe.

`rows() → list[dict[str, Any]]`

flacht die Solverergebnisse für TSV/CSV ab.

`write_tsv(path: str | Path) → Path`

schreibt die Laufdaten.

`evaluate(evaluators: Mapping[str, StandardEvaluator]) → StudyEvaluation`

Wendet standardisierte Auswerter auf die Varianten an.

`StudyEvaluation.rows() → list[dict[str, Any]]`

liefert eine tabellarische Darstellung inklusive Evaluatorfehlern.

7. Standardisierte Evaluatoren

Alle Evaluatoren werden mit einem `EvaluationContext` aufgerufen. Dieser enthält `variant`, `run` und `result_directory`.

`PerformanceEvaluator()`

Solverstatus, Echtzeit, Simulationszeit, Fortschrittsquelle und Performancekennwert. Funktioniert auch bei unvollständigen Ergebnissen.

`MoistureAccumulationEvaluator(threshold_percentage=1.0, from_total=True)`

Feuchteakkumulation und Stabilitätskennwerte.

`FrostRiskEvaluator()`

Frost-Risikobereiche, Anzahl und Flächen.

`MouldRiskEvaluator(material_class=1, surface_class=1, decline_class=4)`

Schimmelbewertung für eindimensionale Projekte. Liefert unter anderem die Zeit über 80 % RH, die Isoplethen-Grenze sowie die VTT-Indizes 1 und 2.

Beispiel:

```
from pydelphin import FrostRiskEvaluator, PerformanceEvaluator

evaluation = result.evaluate({
```

```
    "performance": PerformanceEvaluator(),
    "frost": FrostRiskEvaluator(),
})
rows = evaluation.rows()
```

8. Ergebnis- und Logauswertung

`read_tsv(path: str | Path) → AsciiTable`

Liest eine numerische TSV-Datei. `AsciiTable.headers`, `rows` und `column(name)` stehen für die weitere Analyse zur Verfügung.

`read_d6o(path: str | Path) → D6OResult`

Liest eine DELPHIN-Ausgabedatei.

`read_screenlog(path: str | Path) → list[ScreenLogEntry]`

Liest Solverfortschritte aus der Screenlog. Das ist besonders wichtig, wenn noch keine vollständige Ergebnisdatei geschrieben wurde.

`read_solver_logs(directory: str | Path) → SolverLogs`

Sucht und liest die bekannten Logdateien eines Projektverzeichnisses.

`column_statistics(values: Sequence[float], times: Sequence[float] | None = None) → ColumnStatistics`

Berechnet Grundstatistiken einer Ergebnisreihe; bei vorhandenen Zeitwerten werden auch zeitbezogene Kennwerte berücksichtigt.

9. Assessments

`enable_assessments(project: DelphinProject, *, moisture_accumulation: bool = True, moisture_accumulation_single: bool = False, ice_criteria: bool = False, mould_growth: bool = False) → list[str]`

Aktiviert die ausgewählten Assessments und gibt die vollständige Liste der aktivierten Assessmenttypen zurück.

`annual_values(result: D6OResult, year_days: int = 365) → tuple[list[float], float]`

Extrahiert Jahreswerte und die verfügbare Simulationsdauer.

`assess_moisture_accumulation(output_directory, *, threshold_percentage: float = 1.0, from_total: bool = True, layers: list[int] | None = None) → MoistureAccumulationAssessment`

Bewertet die Feuchteakkumulation aus den internen Ergebnisdateien.

`assess_frost_risk(output_directory) → FrostRiskAssessment`

Bewertet Frostschadensbereiche und deren Flächen.

`assess_mould_risk(output_directory, *, material_class: int = 1, surface_class: int = 1, decline_class: int = 4, last_year_days: int = 365) → MouldAssessment`

Bewertet Schimmelrisiken für die Innenoberfläche eines 1D-Projekts.

Die Assessment-Objekte enthalten neben den Kennwerten auch Angaben dazu, ob die Ergebnisdaten vollständig auswertbar waren. Fehlende oder unpassende Ausgaben werden als `AssessmentDataError` gemeldet.

10. JSON-Konfiguration

Das Script `examples/run_configured_study.py` unterstützt derzeit den Studientyp `material_replacement`. Die JSON-Konfiguration enthält:

- `study_type`
- `project`
- `output_dir`
- `solver`
- `replacement_material`
- `replacement_file`
- `max_workers`
- `solver_run.wall_time_limit_s`
- `solver_run.progress_timeout_s`
- `solver_run.poll_interval_s`
- `solver_run.processes`
- `solver_run.termination_grace_s`
- `evaluators`
- `evaluation_file`
- `reuse_existing`

Pfade werden relativ zum Speicherort der JSON-Datei aufgelöst. Als `evaluators` sind derzeit `performance`, `moisture`, `frost` und `mould` verfügbar.

Aufruf:

```
cd /home/fechner/Delphin6/scripts/PyDelphin
PYTHONPATH=src python3 examples/run_configured_study.py \
    examples/material_replacement_study.json
```

11. Stabilität und Kompatibilität

- Projektdateien werden erst im Speicher geändert und anschließend als neue Datei geschrieben.
- Bei Variantenstudien bleibt das Basisprojekt unverändert.
- Selbstzuweisungen beim Materialtausch werden ignoriert.
- Solver-Timeouts werden als Ergebnisstatus und nicht als verlorene Python-Ausnahme behandelt.
- Nicht jede DELPHIN-Projektstruktur ist bereits durch einen spezialisierten Adapter abgedeckt. Für solche Bereiche kann `DelphinProject.root` zusammen mit den XML-Hilfsfunktionen verwendet werden.