Fehleranalyse
Zirkuläre Importe in der MDP

Problemstellung

In MDP-basierten Szenen hängen Module wie `mdp`, `terminations`, `rewards`, `observations`, `task` und `registry` wechselseitig voneinander ab. Wenn zwei oder mehr Module sich beim Import direkt oder indirekt gegenseitig referenzieren, entsteht ein *zirkulärer Import*. Python lädt dann ein Modul nur teilweise, Objekte sind noch *nicht* definiert, und bereits der Start bricht ab.

Typische Symptome

- `ImportError: cannot import name X from partially initialized module Y`
- `AttributeError: module '...' has no attribute 'ClassOrFunc'`
- Registry bleibt leer, Task wird nicht gefunden, obwohl der Code vorhanden ist.
- Abbruch sehr früh (vor Stage-Load), meist beim ersten Import von `mdp/task`.

Häufige Ursachen

- Wechselseitige Top-Level-Imports zwischen `mdp/__init__.py`, `terminations.py`, `rewards.py`, `task.py`.
- Side-Effects in `__init__.py` (z. B. Registry-Calls), die weitere Imports auslösen.
- Typannotationen mit `from X import Y` auf Top-Level.
- Gemeinsame Konstanten/Typen fehlen; stattdessen greifen Module direkt aufeinander zu.

Minimalbeispiele

mdp/___init___.py

```
from .terminations import is_done # side-effect at package import
from .mdp import PickApplesMDP # pulls mdp.py (back-reference chain)
```

mdp/terminations.py

```
from .mdp import PickApplesMDP # import back to mdp.py -> cycle
def is_done(state):
    ...
```

mdp/mdp.py

```
from .terminations import is_done # again back to terminations.py
class PickApplesMDP:
    ...
```

Fix-Patterns

Pattern A: Ebenen entflechten

- **Leaf-Module**: `terminations.py`, `rewards.py`, `observations.py` dürfen *nicht* zu `mdp.py` oder `task.py` zurückimportieren.
- Gemeinsame Konstanten/Typen in `common.py` oder `types.py` bündeln.

Pattern B: Single Source of Truth

- `registry.py` registriert Tasks an genau *einer* Stelle.
- `__init__.py` nur Exporte, *keine* Side-Effects (keine Registry-Calls, kein Import-Fächern).

Pattern C: Lazy-Imports

- Optionale/seltene Imports in Funktionskörper verschieben.
- Späte Bindung statt Top-Level-Import, wenn nur eine Funktion benötigt wird.

Pattern D: TYPE_CHECKING für Typannotationen

- Import für Typen schützen:

```
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from .mdp import PickApplesMDP
```

Pattern E: Schnittstellen definieren

- Statt direktem Funktionsimport (`from .terminations import is_done`) nur *Protokoll* erwarten, z. B. Callables im Konstruktor injizieren.

Beispiel: Entzerrte Struktur

mdp/common.py

```
from dataclasses import dataclass

@dataclass
class MDPParams:
    max_steps: int = 200
```

mdp/terminations.py

```
def is_done_default(state) -> bool:
    return state["t"] >= state["params"].max_steps
```

mdp/mdp.py

```python
from .common import MDPParams

class PickApplesMDP:
    def __init__(self, is_done):
        self.params = MDPParams()
        self.is_done = is_done # injected callable
```

task.py

```python
from mdp.mdp import PickApplesMDP
from mdp.terminations import is_done_default

def build_task():
    mdp = PickApplesMDP(is_done=is_done_default)
    return mdp
```

Refactoring-Strategien

- **Top-Level-Imports reduzieren:** Alles, was nicht absolut nötig ist, in Funktionen verschieben.
- **Abhängigkeitsrichtung festlegen:** `mdp` darf `terminations`/`rewards` konsumieren, aber diese *nicht* `mdp`.
- **Factories einsetzen:** `build\_task()` in `task.py` konstruiert Objekte und verkabelt Abhängigkeiten.
- **Wrapper/Adapter:** Unterschiede zwischen Versionen/Modulen zentral abfangen, nicht quer referenzieren.

Debug-Playbook

1. **Stacktrace lesen:** Erstes `partially initialized module` lokalisieren.
2. **Importgraph skizzieren:** Wer importiert wen auf Top-Level? Rückkante markieren.
3. **Probe-Imports:** In einer Minimal-Shell nacheinander `import mdp.terminations`, `import mdp.mdp`.
4. **Zyklische Kante brechen:** Einen der beiden Top-Level-Imports in eine Funktion verschieben.
5. **TYPE_CHECKING anwenden:** Nur Typen *unter* `if TYPE\_CHECKING:` importieren.
6. **Side-Effects entfernen:** Registry-Aufrufe aus `__init__.py` verbannen.
7. **Caches löschen** und sauber neu starten:

```
find . -type d -name "__pycache__" -exec rm -rf {} +
rm -rf .pytest_cache .cache outputs
rm -rf ~/.cache/ov ~/.local/share/ov/Kit*
```

Tests und Guards

```python
# tests/test_imports.py
def test_import_order():
    import importlib
    importlib.invalidate_caches()
    importlib.import_module("mdp.terminations")
    importlib.import_module("mdp.mdp")

# tests/test_registry.py (falls Registry genutzt wird)
from core.registry import task_registry
from tasks.pick_apples.registry import register, TASK_NAME

def test_task_registered_once():
    task_registry.clear()
    register(task_registry)
    keys = [k for k in task_registry.keys() if k == TASK_NAME]
    assert len(keys) == 1
```

Quick-Checklist

- Keine wechselseitigen Top-Level-Imports zwischen `mdp`, `terminations`, `rewards`, `task`.
- `__init__.py` ohne Side-Effects, Registry nur zentral.
- Typen-Imports hinter `if TYPE\_CHECKING:`.
- Gemeinsame Konstanten in `common.py`, nicht quer importieren.
- Nach Änderungen Caches löschen, Prozess neu starten.