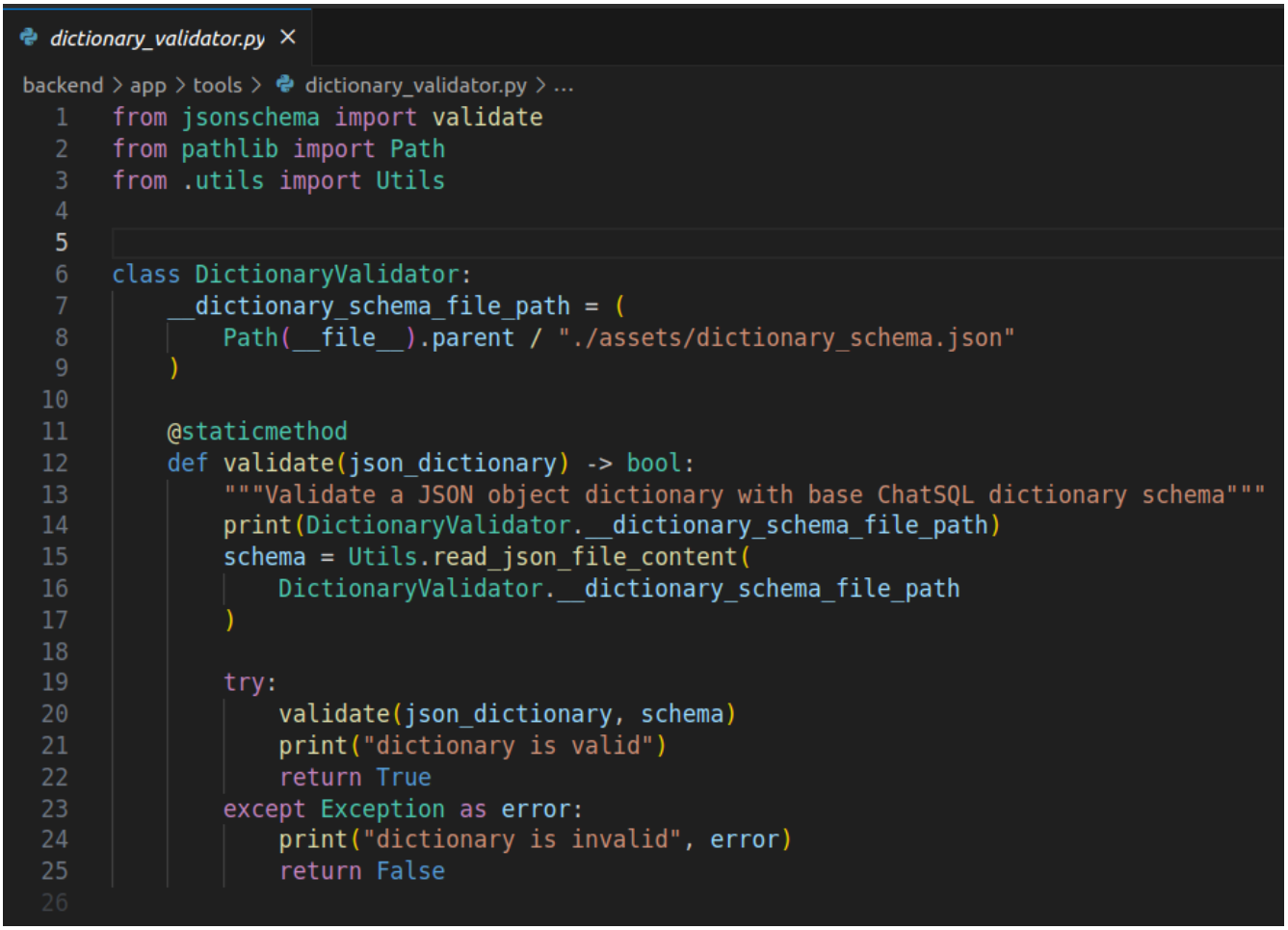
Come usare Pytest

partendo dal modulo dictionary_validator.py



Prima dobbiamo capire chi è la classe DictionaryValidator. Questo vuol dire capire quali sono le funzionalità chiave e il loro workflow, senza preoccuparci delle comunicazioni effettive con altri file.

Analisi del modulo

Primo passo: *@staticmethod*

Al centro della classe c'è un *metodo statico*.
Vuol dire che è legato alla classe, non ad un'istanza della classe. Quindi può essere chiamato sulla classe stessa, e non su una specifico oggetto istanziato dalla classe. Non ha accesso alla keyword *self,* e di conseguenza non può alterare o accedere alle variabili di istanza.
Nella logica dei test di unità, i metodi statici sono più semplici da testare proprio perché non si deve creare nessuna istanza di classe. Quindi per testarli basta chiamarli con i parametri voluti, e il loro comportamento resta consistente per ogni utilizzo ammesso che i parametri siano sempre gli stessi.

Secondo passo: *Path del file dizionario dati*
La classe viene inizializzata con un path al dizionario dati in JSON. Quindi stiamo parlando di una *dipendenza esterna*.

Terzo passo: *Lettura del file dal suo path*
Dentro il metodo statico si va a leggere il dizionario dati chiamando una funzione da *Utils*. Questa si deve occupare della lettura e parsing del file JSON in un dizionario secondo la sintassi di Python.

Quarto passo: *Validazione del file*
Dalla classe *jsonschema* usiamo la funzione *validate* per validare l'input in JSON del dizionario dati contro lo schema. Siamo in una try, quindi se la validazione fallisce viene lanciata un'eccezione che viene catturata dalla *validate* che stiamo definendo come metodo statico.

Quinto passo: *Gestione delle eccezioni e output*
Ogni eccezione sollevata viene accolta, si ha l'output per come definito e ritorna False.
Se invece la validazione ha successo, si ha un ok e ritorna True.

Questo serve a capire come è strutturata la classe e come funziona il suo metodo principale (e unico in questo caso).

Per creare dei test di unità, vanno considerate tre cose:

1. **Isolamento**
   Il metodo deve essere testato in isolamento dai file esterni e dalla logica di validazione data da *jsonschema*.

2. **Verifica dei risultati ritornati**
   Bisogna verificare i valori ritornati e i loro side effects.

3. **Verifica della gestione degli errori**
   Bisogna verificare come vengono gestiti gli errori aspettati e quelli inaspettati.

Identificazione dei mock
Dato che il modulo deve essere isolato, le dipendenze e i collegamenti con tutto quello che sta fuori dalla classe deve essere simulato, o mockato. Quindi bisogna identificare le dipendenze, come abbiamo fatto sopra, e pensare a come creare dei mock. Rivediamo le dipendenze nel dettaglio:

1. **Utils.read_json_file_content**
   Questa funzione si occupa della lettura di un file => Deve essere mockata per evitare una lettura reale.

2. **jsonschema.validate**
   Questa funzione serve a capire la validità del dizionario dati contro uno schema, e il suo comportamento è lanciare eccezioni o dare un via libera.

Concretamente però, come si fanno i mock?
Con Pytest possiamo crearli col comando *mock*, che vediamo dopo.

Per *Utils.read_json_file_content*, possiamo controllare lo schema ritornato durante i test, e simulare anche schemi corrotti o invalidi.
Per *jsonschema.validate*, possiamo configurare il mock per sollevare un'eccezione o simulare una validazione con successo.

Ragionare su cosa vogliamo verificare coi test

Prima di mettersi a scrivere dei test a caso, è meglio pensare a cosa abbia senso verificare.
Conviene sempre dividere in test per input validi, per vedere che succede avvicinandoci a situazioni limite, e test per input non validi. Agli input non validi seguono i test per la corretta gestione degli errori.

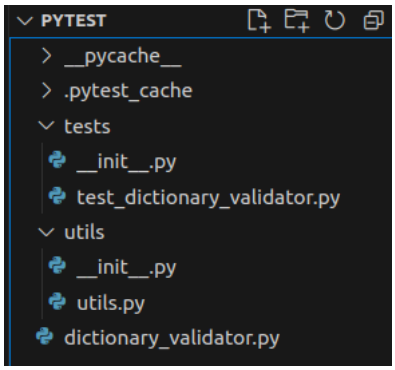
1. Test per input validi:
   - Casi normali: Un dizionario che matcha con lo schema
   - Casi limite: Il minimo input valido, il massimo input valido
2. Test per input non validità:
   - Dizionari che violano lo schema (campi mancanti ecc.)
3. Test per la gestione degli errori:
   - Simulare errori di lettura (file mancante, accesso al file negato)
   - Simulare errori di parsing (JSON invalido)

Si può anche verificare se tutte le possibili eccezioni siano gestite individualmente:

4. Test per le eccezioni:
   - Configurare il mock di jsonschema.validate per lanciare diverse eccezioni e vedere se vengono gestite correttamente.

Creare i test con Pytest

richiede un certo setup.
Prima, serve una buona struttura delle directory:



Inoltre, è necessario installare pytest e pytest-mock.

pip install pytest pytest-mock

Partiamo capendo cosa importare.

1. Il modulo da testare
2. Pytest
3. La libreria per gestire i path da cui prendiamo il JSON
4. Un modo per testare la gestione degli errori ed eccezioni

Per i tre sono i soliti import:
from dictionary_validator import DictionaryValidator
import pytest #importante la minuscola!
from pathlib import Path

Per la gestione degli errori, bisogna gestire gli errori relativi al JSON in input. La libreria jsonschema fornisce ValidationError, che viene sollevata dal metodo validate. Quindi:
from jsonschema.exceptions import ValidationError

Primo passo:

Definiamo cosa testare di *DictionaryValidator*. La funzionalità di base è la validazione di un JSON correttamente formattato secondo uno schema predeterminato.

Secondo passo:

Vanno determinati gli input e gli output del test.
Ad esempio, per il metodo *validate*, possiamo pensare a:
Input: Uno schema JSON valido
Output: True

Terzo passo:

Vogliamo vedere se il metodo *validate* (quello definito nella classe, non quello della libreria) funziona. Il problema è che si affida al metodo *validate* di *jsonschema* e al medoto *read_json_file_content* di *Utils*. Questi due metodi vanno mockati (a meno che io non voglia usare il vero file utils).
Qua, in effetti, si può usare il vero file *utils.py*. Nonostante sia un modulo che può essere completamente mockato, averlo reale può servire in futuro per i test di integrazione, che assicurano che le componenti funzionino bene insieme. Se poi *utils* viene usato da altre parti del codice, è bene che sia implementato per come lo è veramente. Adesso però lo mockiamo per imparare a farlo.

Prima di andare avanti bisogna capire concretamente cosa sia il **mocking**.
Per definizione, è una tecnica usata nei test di unità per simulare il comportamento di oggetti reali complessi. Sostituendoli con dei mock che simulano il loro comportamento, possiamo isolare completamente i moduli che vogliamo testare, rimuovere le dipendenze esterne o elementi imprevedibili, e specificare condizioni o comportamenti che sarebbero difficili da riprodurre con gli oggetti effettivi, come raggiungere specifici errori o eccezioni.

Nel caso di *DictionaryValidator*, è importante isolare il metodo statico *validate* e controllare gli input e comportamenti delle dependencies esterne senza stare dietro a reali operazioni di I/O.
Vediamo come.

- **Utils.read_json_file_content**

Legge uno schema JSON dal filesystem. Deve ritornare un oggetto JSON che rappresenta uno schema. Quando *DictionaryValidator.validate* chiama questo metodo, non dovrà quindi leggere da un file da qualche parte nel computer, ma ottenere uno schema qualunque che rispetti le regole del JSON.
Pytest permette, con *mocker.patch*, di prendere la nostra funzione o metodo che vogliamo mockare, e sostituirla col mock. Prima definiamo il mock, cioè lo schema che sappiamo che va bene perché rispetta le regole del nostro dizionario dati. **Qua userò uno schema banale per evitare un codice lunghissimo**:

mock_schema = {"type": "object", "properties": {"key": {"type": "string"}}}

Poi lo diamo come quello che viene ritornato a priori dalla chiamata a *Utils.read_json_file_content*:

mocker.patch('utils.utils.Utils.read_json_file_content', return_value=mock_schema)

Così, quando viene chiamato quel metodo, si effettua il ritorno di quello impostato come return_value bypassando tutta la logica del metodo definita altrove.

- **jsonschema.validate**

Questa funzione serve a validare un oggetto JSON contro il solito schema JSON. Cioè, se del JSON rispetta o no la costruzione di un file JSON. Se ha successo, non produce alcun output; altrimenti, solleva un ValidationError.
Bisogna quindi simulare questi due comportamenti di successo e fallimento.

1. In caso di successo:
   mocker.patch('jsonschema.validate')

2. In caso di fallimento:
   mocker.patch('jsonschema.validate', side_effect=ValidationError("Invalid JSON"))

La prima funzione sta prendendo jsonschema.validate e sostituendola con un mock che non fa assolutamente nulla: nella pratica, coincide nel caso in cui non si specifica cosa debba ritornare (non è specificato un return_value) o se deve comportarsi secondo un side effect (non è specificato side_effect). Quindi è valida per essere il mock del caso in cui le cose vanno bene.

La seconda funzione invece prende jsonschema.validate e la sostituisce con un mock che ritorna un side effect (che è diverso da un normale return), cioè un ValidationError che comunica che il JSON non è valido. Viene sollevata come eccezione, quindi non poteva essere un return normale.

Terzo passo e mezzo:

Questi mock non vanno inseriti nel codice come se fossero metodi di classe, ma vengono definiti *dentro il test*. Quindi è bene magari appuntarseli e specificare quale mock serve a quale casistica.

Quarto passo:

I mock sono stati definiti, quindi si possono creare i test. Un test verifica una cosa specifica, quindi, nel nostro caso, possiamo vedere come si comporta il metodo statico *validate* nei casi di successo e fallimento. Servono due test: *test_validate_successful* e *test_validate_failure_due_to_invalid_json.*
Per i nomi, si parte con test, poi il nome della funzione e infine il caso che stiamo verificando.
Comunque guardiamo solo il primo, perché il secondo caso è analogo.

test_validate_successful

Il codice risulta poco leggibile perché lo schema è enorme, ma basta capirne il senso:

```
def test_validate_successful(mocker):
    # Il valid_json è del JSON per cui andiamo a chiedere la verifica e che sappiamo già essere valido
    valid_json = {
        "database_name": "SampleDB",
        "database_description": "A sample database schema.",
        "tables": [
            {
                "name": "table1",
                "description": "A sample table",
                "table_synonyms": ["tbl1", "t1"],
                "columns": [
                    {
                        "name": "column1",
                        "type": "varchar",
                        "description": "A sample column",
                        "column_synonyms": ["col1", "c1"]
                    }
                ],
                "primary_key": ["column1"],
                "foreign_keys": [
                    {
                        "foreign_key_column_names": ["column1"],
                        "reference_table_name": "table2",
                        "reference_column_names": ["column2"]
                    }
                ]
            }
        ]
    }

    # Il nostro schema JSON
    mock_schema = {
    "type": "object",
    "required": [
        "database_name",
        "database_description",
        "tables"
    ],
    "properties": {
```

```
"database_name": {
    "type": "string"
},
"database_description": {
    "type": "string"
},
"tables": {
    "type": "array",
    "minItems": 1,
    "items": {
        "type": "object",
        "required": [
            "name",
            "description",
            "columns",
            "primary_key"
        ],
        "properties": {
            "name": {
                "type": "string"
            },
            "description": {
                "type": "string"
            },
            "table_synonyms": {
                "type": [ "array", "null" ],
                "items": {"type": "string"}
            },
            "columns": {
                "type": "array",
                "minItems": 1,
                "items": {
                    "type": "object",
                    "required": [
                        "name",
                        "type",
                        "description"
                    ],
                    "properties": {
                        "name": {
                            "type": "string"
                        },
                        "type": {
                            "type": "string"
                        },
                        "description": {
                            "type": "string"
                        },
                        "column_synonyms": {
```

```
                        "type": [ "array", "null" ],
                        "items": {"type": "string"}
                    }
                }
            }
        },
        "primary_key": {
            "type": "array",
            "items": {"type": "string"},
            "minItems": 1
        },
        "foreign_keys": {
            "type": "array",
            "items": {
                "type": "object",
                "required": [
                    "foreign_key_column_names",
                    "reference_table_name",
                    "reference_column_names"
                ],
                "properties": {
                    "foreign_key_column_names": {
                        "type": "array",
                        "items": {"type": "string"}
                    },
                    "reference_table_name": {
                        "type": "string"
                    },
                    "reference_column_names": {
                        "type": "array",
                        "items": {"type": "string"}
                    }
                }
            }
        }
    }
}

    # Ora mettiamo il mock della lettura del JSON di Utils
    mocker.patch('utils.utils.Utils.read_json_file_content', return_value=mock_schema)
    # Poi il mock di jsonschema.validate, che non fa nulla
    mocker.patch('jsonschema.validate')

    # E il test effettivo sono queste due righe
    result = DictionaryValidator.validate(valid_json)
    assert result is True, "The validator should return True for valid JSON input"
```

Concludiamo vedendo le ultime due righe.

- **result = DictionaryValidator.validate(valid_json)**

Si ha una chiamata al metodo che vogliamo testare.
L'input che gli passiamo è *valid_json*, che è quello che vogliamo che venga validato.
L'output è un booleano, e viene salvato nella variabile *result*.
Quello che accade quindi è catturare l'output della validazione su qualcosa che sappiamo che ci deve dare True.

- **assert result is True, "The validator should return True for valid JSON input"**

assert è un comando che verifica se una condizione è True. Se lo è, bene, il test è passato; se è False, Pytest riporta gli errori. Si lancia su *result*, e appunto si verifica che sia True.
In seguito si ha un messaggio d'errore, dicendo cosa debba essere restituito in caso il *result* sia False.