pydantic

August 20, 2025

1 Pydantic

Pydantic to biblioteka służąca do tworzenia i walidacji modeli struktur danych czyli schematów danych o strukturze typu JSON.

Przykładowe **zastosowania** modeli pydantica: - definiowanie request body w REST API - definiowanie formatu formatu JSONa według którego odpowiedź ma wygenerować LLM - walidacja poprawności formatu danych JSON - zamiana danych w postaci słownika na instancję klasy modelu (możemy wtedy np. zdefiniować dla takich obiektów odpowiednie metody)

1.1 `BaseModel` i modele danych

[ ]:
```python
from pydantic import BaseModel
```

[ ]:
```python
class User(BaseModel):
    username: str
    password: str
    is_admin: bool
```

[ ]:
```python
class Task(BaseModel):
    description: str
    priority: int
    is_completed: bool
    assigned_user: User
```

1.2 Walidacja JSONa

Do walidacji JSONa względem modelu używamy metody `model_validate()`.

Poprawna walidacja

W przypadku kiedy słownik pasuje do modelu, `model_validate()` zwraca instancję tego modelu.

[ ]:
```python
user = {"username": "John Doe", "password": "my_password", "is_admin": True}
task = {"description": "Task description", "priority": 2, "is_completed":␣
 ↪False, "assigned_user": user}
```

[ ]:
```python
User.model_validate(user)
```

[ ]:
```python
Task.model_validate(task)
```

Poniżej `priority` jest zapisane jako `str`. Ponieważ jednak da się to zrzutować na `int`, to walidacja również przebiega poprawnie.

[ ]:
```
user = {"username": "John Doe", "password": "my_password", "is_admin": True}
task = {"description": "Task description", "priority": "2", "is_completed":␣
 ↪False, "assigned_user": user}
```

[ ]:
```
Task.model_validate(task)
```

Niepoprawna walidacja

W przypadku błędu walidacji dostajemy błąd `ValidationError`.

[ ]:
```
user = {"username": "John Doe", "password": "my_password", "is_admin": True}
task = {"description": "Task description", "priority": "invalid type",␣
 ↪"is_completed": False, "assigned_user": user}
```

[ ]:
```
Task.model_validate(task)
```

1.3 Rzutowanie słownika na obiekt

Rzutowanie słownika na obiekt pozwala nie tylko przechowywać informacje w postaci tego obiektu zamiast słownika ale również korzystać ze wszystkich metod odpowiadającej mu klasy.

[ ]:
```
user
```

[ ]:
```
User(**user)
```

[ ]:
```
class User(BaseModel):
    username: str
    password: str
    is_admin: bool

    def change_password(self, new_password):
        if isinstance(new_password, str):
            self.password = new_password
        else:
            raise TypeError("Invalid type for new password")
```

[ ]:
```
user_obj = User(**user)
user_obj
```

[ ]:
```
user_obj.change_password("new_password")
user_obj
```

[ ]:
```
user_obj.change_password(True)
```

ZADANIE

Stwórz klasę `Task`, która będzie miała następujące pola (atrybuty). Dobierz dla nich odpowiednie typy. - `description` - `assigned_user` - `due_date` - `comments`

Klasa ta powinna również posiadać metodę `modify_description()`, której zadaniem będzie modyfikacja opisu zadania. Jeśli podano wartość typu innego niż `str` należy zwrócić odpowiedni błąd.

Następnie na podstawie słownika z danymi o zadaniu dokonaj walidacji i zamiany słownika na obiekt.

[ ]:
```
# ...
```

1.4 Zaawansowane elementy definiowania modeli

Dopuszczanie różnych typów

[ ]:
```
class Task(BaseModel):
    description: str
    priority: int | str
    is_completed: bool | None
```

[ ]:
```
Task(description="task description", priority="low", is_completed=None)
```

[ ]:
```
class Task(BaseModel):
    description: str
    priority: float  # | int
    is_completed: bool | None
```

[ ]:
```
Task(description="task description", priority=2, is_completed=False)
```

Typ `Literal`

[ ]:
```
from typing import Literal
```

[ ]:
```
class Task(BaseModel):
    description: str
    priority: Literal["low", "medium", "high"]
    is_completed: bool
```

[ ]:
```
Task(description="task description", priority="medium", is_completed=True)
```

Typy z ograniczeniami

Constrained types

[ ]:
```
from pydantic import constr, conint, confloat
```

[ ]:
```python
class Test(BaseModel):
    test_string: constr(min_length=5, max_length=10, pattern=r"^[a-zA-Z]+$")
    test_int: conint(gt=2, lt=8)
    test_float: confloat(ge=0.0, le=1.0)
```

[ ]:
```python
test = Test(test_string="hello", test_int=3, test_float=0.5)
test
```

Wartości domyślne i opcjonalne

[ ]:
```python
from datetime import date
```

[ ]:
```python
class User(BaseModel):
    username: str
    password: str
    is_admin: bool = False
    date_of_birth: date | None = None
```

[ ]:
```python
User(username="John Doe", password="password")
```

Field – metadane pola

`Field` pozwala zdefiniować wiele rzeczy w jednym miejscu, m.in. wartość domyślną, walidację wartości, alias pola czy jego opis.

[ ]:
```python
from pydantic import Field
from datetime import datetime
```

[ ]:
```python
class User(BaseModel):
    username: str = Field(alias="login")
    password: str = Field(min_length=8)
    is_admin: bool = Field(False, description="Whether or not the user is an␣
 ↪admin")
    creation_timestamp: date = Field(default_factory=datetime.now)
```

[ ]:
```python
User(login="my_username", password="my_password")
```

`field_validator` i `model_validator` – customowa walidacja

Jeśli chcemy zastosować bardziej zaawansowaną walidację modelu możemy użyć metod z dekoratorami `field_validator` lub `model_validator`.

`field_validator` - służy do walidacji wartości w pojedynczym polu

`model_validator` - służy do walidacji zależności między polami modelu

[ ]:
```python
from pydantic import field_validator, model_validator
```

[ ]:
```python
class User(BaseModel):
    username: str = Field(alias="login")
    password: str = Field(min_length=8)
```

```
    is_admin: bool = Field(False, description="Whether or not the user is an␣
↪admin")
    creation_timestamp: date = Field(default_factory=datetime.now)

    @field_validator("password")
    def check_password_not_contains_admin(cls, value):
        if "admin" in value.lower():
            raise ValueError("Password should not contain the word 'admin'")
        return value

    @model_validator(mode="after")
    def validate_password_length_for_admins(self):
        min_admin_password_length = 12
        if self.is_admin and len(self.password) < min_admin_password_length:
            raise ValueError("Too short password for an admin")
        return self
```

[ ]:
```
User(login="user", password="user1234")  # admin1234
```

[ ]:
```
User(login="user", password="user12345678", is_admin=True)  # user1234
```

ZADANIE

Rozbuduj klasę `Task` z poprzedniego zadania w następujący sposób: - w polu `assigned_user` zezwól na wartość typu `str` (username użytkownika) - dodaj pole `category`, które będzie miało kilka określonych wartości dopuszczalnych - ustaw maksymalną długość opisu na 30 znaków - przyjmij pustą listę jako domyślną wartość pola `comments` - dodaj opis wybranego pola korzystając z `Field`

Następnie stwórz instancję tej klasy.

[ ]:
```
# ...
```

1.5 Pydantic + LLM

Poniższego kodu **nie należy** wykonywać (jest do tego potrzebny klucz API OpenAI). Obrazuje on jednak w jaki sposób możemy zastosować Pydantica w pracy z modelami językowymi.

[ ]:
```
from openai import OpenAI
```

[ ]:
```
client = OpenAI()
```

[ ]:
```
class User(BaseModel):
    username: str = Field(description="Username of the user")
    password: str = Field(description="Password of the user")
    is_admin: bool = Field(description="Whether or not the user is an admin")
```

[ ]:
```
response = client.responses.parse(
    model="gpt-4o",
```

```
    input=[
        {
            "role": "user",
            "content": "Generate a sample user for test purposes",
        },
    ],
    text_format=User,
)
```

[ ]:
```
response.output[0].content[0].text
```

```
'{"username":"test_user123","password":"secureP@ssw0rd!","is_admin":false}'
```

[ ]:
```
response.output_parsed
```

```
User(username='test_user123', password='secureP@ssw0rd!', is_admin=False)
```