Panda Dataframe, scypy, numpy, etc.

Md Rasheduzzaman

2025-09-09

Pydantic

Table of contents

1 Dynamic dtype

Data typing in python: It is dynamic in python. But we can put some hints to help users with their input. But still, it can be problematic. See below:

```
def insert_patient_data(name: str, age: int):
  print(name)
  print(age)
  print("inserted into the DB")

insert_patient_data("Rashed", "thirty")
```

```
Rashed
thirty
inserted into the DB
```

You see, nobody is stopping the user to put age as a string. A better way would be to keep a check on the data type using loop. If the data type doesn't match, we will raise an error.

```
def insert_patient_data(name: str, age: int):
  if type(name)==str and type(age)==int:
    print(name)
```

```
    print(age)
    print("inserted into the DB")
  else:
    raise TypeError("Incorrect data type")

insert_patient_data("Rashed", 30)
```

```
Rashed
30
inserted into the DB
```

```
insert_patient_data("Rashed", "thirty")
```

```
TypeError: Incorrect data type
```

But this way is not scalable.

```
def insert_patient_data(name: str, age: int):
  if type(name)==str and type(age)==int:
    print(name)
    print(age)
    print("inserted into the DB")
  else:
    raise TypeError("Incorrect data type")

def update_patient_data(name: str, age: int):
  if type(name)==str and type(age)==int:
    print(name)
    print(age)
    print("Updated")
  else:
    raise TypeError("Incorrect data type")

insert_patient_data("Rashed", 30)
```

```
Rashed
30
inserted into the DB
```

```
update_patient_data("Rashed", 30)
```

```
Rashed
30
Updated
```

You see the issue with scalability? How many times will we do it if we have more functions using these variables? Data validation is also very important for us for better control. In the above example, we could put `-10` as age, it would pass the data type check, there is no stopping. But is it meaningful? So, we could say `age` can not be less than 0. How to do it?

```
def insert_patient_data(name: str, age: int):
  if type(name)==str and type(age)==int:
    if age < 0:
      raise ValueError("Age cannot be less than 0")
    else:
      print(name)
      print(age)
      print("Inserted into the DB")
  else:
    raise TypeError("Incorrect data type")
```

Now, let's check.

```
insert_patient_data("Rashed", 10)
```

```
Rashed
10
Inserted into the DB
```

```
insert_patient_data("Rashed", -10)
```

```
ValueError: Age cannot be less than 0
```

```
insert_patient_data("Rashed", "10")
```

```
TypeError: Incorrect data type
```

Here comes `Pydantic` to help us checking for

- Data type, and
- Data validation

And it does so in 3 steps:
- 1. **Define a Pydantic model (class)** representing the **ideal schema**. This includes the expected fields, their data types and any validation constraint (e.g. `lt=0` for negative numbers) - 2. **Instantiate the model with raw input data** or make a Pydantic object (usually a dictionary or JSON-like structure) - Pydantic will automatically **validate** the data and **coerce** it into the correct Python types (if possible) - If the data doesn't meet the model's criteria, Pydantic raise a `ValidationError`. - 3. Pass the validated model object to functions or use it throughout your codebase. - This ensures that every part of your program works with **clean, type-safe, and logically valid data**.

Let's use it now. But let's make the example more realistic. We will make a dataframe with the required fields using pandas first. Then we will insert a patient info into that dataframe if the patient is new. If not, we will update information for that patient.

```
from pydantic import BaseModel, ValidationError
import pandas as pd

# ---------------------
# 1. Define the model
# ---------------------
class Patient(BaseModel):
    name: str
    age: int
    weight: float

# ---------------------
# 2. In-memory database
# ---------------------
# Create a DataFrame to store patient records
db = pd.DataFrame({
    'name': pd.Series(dtype='str'),
    'age': pd.Series(dtype='int'),
    'weight': pd.Series(dtype='float')
})

# ---------------------
# 3. Insert function
# ---------------------
def insert_patient_data(patient: Patient):
    global db
```

```
    # Check if patient already exists by name
    if db['name'].eq(patient.name).any():
        print(f"Patient '{patient.name}' already exists. Use update instead.")
        return

    # Append new patient
    db = pd.concat([db, pd.DataFrame([patient.model_dump()])], ignore_index=True)
    print(f"Inserted patient: {patient.name}")

# ---------------------
# 4. Update function
# ---------------------
def update_patient_data(patient: Patient):
    global db
    # Find index of the patient by name
    idx = db.index[db['name'] == patient.name].tolist()
    if not idx:
        print(f"Patient '{patient.name}' not found. Use insert instead.")
        return

    # Update the record
    db.loc[idx[0], ['age', 'weight']] = patient.age, patient.weight
    print(f"Updated patient: {patient.name}")

# ---------------------
# 5. Test the system
# ---------------------
# Initial insert
patient_info = {'name': 'Rashed', 'age': 29, 'weight': '55'}
try:
    patient1 = Patient(**patient_info) #unpacking using 2 star signs
    insert_patient_data(patient1)
except ValidationError as e:
    print("Validation Error:", e)
```

```
Inserted patient: Rashed
```

```
# Try to insert again (should warn)
insert_patient_data(patient1)
```

```
Patient 'Rashed' already exists. Use update instead.
```

```
# Update patient
updated_info = {'name': 'Rashed', 'age': 30, 'weight': 57.5}
try:
    patient1_updated = Patient(**updated_info)
    update_patient_data(patient1_updated)
except ValidationError as e:
    print("Validation Error:", e)
```

```
Updated patient: Rashed
```

```
# Show database
print("\nCurrent Database:")
```

```
Current Database:
```

```
print(db)
```

```
     name  age  weight
0  Rashed   30    57.5
```

Did you notice something? We put `'weight': '55'` and PyDantic coerced it to float smartly.

But we have another practical issue remaining. Names are not reliable identifier, multiple patients could have the same name. So, we need to handle it correctly using a patient id.

```
from pydantic import BaseModel, ValidationError
import pandas as pd

# ---------------------
# 1. Patient model with manual ID
# ---------------------
class Patient(BaseModel):
    patient_id: str
    name: str
    age: int
    weight: float
```

```python
# ---------------------
# 2. In-memory DB
# ---------------------
db = pd.DataFrame({
    'patient_id': pd.Series(dtype='str'),
    'name': pd.Series(dtype='str'),
    'age': pd.Series(dtype='int'),
    'weight': pd.Series(dtype='float')
})

# ---------------------
# 3. Insert function
# ---------------------
def insert_patient_data(patient: Patient):
    global db
    if db['patient_id'].eq(patient.patient_id).any():
        print(f"Patient ID '{patient.patient_id}' already exists. Use update instead.")
        return
    new_row = pd.DataFrame([patient.model_dump()])
    db = pd.concat([db, new_row], ignore_index=True)
    print(f"Inserted patient: {patient.name} with ID: {patient.patient_id}")

# ---------------------
# 4. Update function
# ---------------------
def update_patient_data(patient: Patient):
    global db
    idx = db.index[db['patient_id'] == patient.patient_id].tolist()
    if not idx:
        print(f"Patient ID '{patient.patient_id}' not found. Use insert instead.")
        return
    db.loc[idx[0], ['name', 'age', 'weight']] = patient.name, patient.age, patient.weight
    print(f"Updated patient: {patient.name} with ID: {patient.patient_id}")

# ---------------------
# 5. Test it
# ---------------------
try:
    # Add 2 patients manually
    patient1 = Patient(patient_id='P001', name='Rashed', age=29, weight=55)
    patient2 = Patient(patient_id='P002', name='Rashed', age=40, weight=70)
```

```
    insert_patient_data(patient1)
    insert_patient_data(patient2)

    # Attempt duplicate insert
    insert_patient_data(patient1)

    # Update patient1
    patient1_updated = Patient(patient_id='P001', name='Rashed', age=30, weight=56.5)
    update_patient_data(patient1_updated)

except ValidationError as e:
    print("Validation Error:", e)
```

```
Inserted patient: Rashed with ID: P001
Inserted patient: Rashed with ID: P002
Patient ID 'P001' already exists. Use update instead.
Updated patient: Rashed with ID: P001
```

```
# ---------------------
# 6. Show DB
# ---------------------
print("\nCurrent Database:")
```

```
Current Database:
```

```
print(db)
```

```
  patient_id    name  age  weight
0       P001  Rashed   30    56.5
1       P002  Rashed   40    70.0
```

Let's make a bit more complex model. We are going to add more fields having more than one entry. So, pandas dataframe is not a good choice. We will use json data format instead.

```
from pydantic import BaseModel, ValidationError
from typing import List, Dict
import json

# ---------------------
```

```python
# 1. Patient model
# ---------------------
class Patient(BaseModel):
    patient_id: str
    name: str
    age: int
    weight: float
    married: bool
    allergies: List[str]
    contact_info: Dict[str, str]

# ---------------------
# 2. In-memory "DB"
# ---------------------
db: List[Patient] = []

# ---------------------
# 3. Insert function
# ---------------------
def insert_patient_data(patient: Patient):
    global db
    if any(p.patient_id == patient.patient_id for p in db):
        print(f"Patient ID '{patient.patient_id}' already exists. Use update instead.")
        return
    db.append(patient)
    print(f"Inserted patient: {patient.name} with ID: {patient.patient_id}")

# ---------------------
# 4. Update function
# ---------------------
def update_patient_data(patient: Patient):
    global db
    for idx, p in enumerate(db):
        if p.patient_id == patient.patient_id:
            db[idx] = patient
            print(f"Updated patient: {patient.name} with ID: {patient.patient_id}")
            return
    print(f"Patient ID '{patient.patient_id}' not found. Use insert instead.")

# ---------------------
# 5. Save/Load to/from JSON
# ---------------------
```

```python
def save_db_to_json(filepath="patients.json"):
    with open(filepath, 'w') as f:
        json.dump([p.model_dump() for p in db], f, indent=2)
    print("Database saved to JSON.")

def load_db_from_json(filepath="patients.json"):
    global db
    try:
        with open(filepath, 'r') as f:
            data = json.load(f)
            db = [Patient(**p) for p in data]
        print("Database loaded from JSON.")
    except FileNotFoundError:
        print("No existing database found.")
    except ValidationError as e:
        print("Validation error while loading:", e)

# ---------------------
# 6. Test it
# ---------------------
try:
    load_db_from_json()

    patient1 = Patient(
        patient_id='P001',
        name='Rashed',
        age=29,
        weight=55,
        married=True,
        allergies=['Dust', 'Pollen'],
        contact_info={'phone': '+492648973', 'email': 'abcrashed@gmail.com'}
    )

    patient2 = Patient(
        patient_id='P002',
        name='Rashed',
        age=40,
        weight=70,
        married=True,
        allergies=['Pollen'],
        contact_info={'phone': '+49663882', 'email': 'rashed@gmail.com'}
```

```
    )

    insert_patient_data(patient1)
    insert_patient_data(patient2)
    insert_patient_data(patient1)  # Duplicate test

    # Update
    patient1_updated = Patient(
        patient_id='P001',
        name='Rashed',
        age=30,
        weight=56.7,
        married=True,
        allergies=['Dust', 'Pollen'],
        contact_info={'phone': '+492648973', 'email': 'abcrashed@gmail.com'}
    )
    update_patient_data(patient1_updated)

    save_db_to_json()

except ValidationError as e:
    print("Validation Error:", e)
```

```
Database loaded from JSON.
Patient ID 'P001' already exists. Use update instead.
Patient ID 'P002' already exists. Use update instead.
Patient ID 'P001' already exists. Use update instead.
Updated patient: Rashed with ID: P001
Database saved to JSON.
```

```
# ----------------------
# 7. Show database
# ----------------------
print("\nCurrent Database (in-memory):")
```

```

Current Database (in-memory):
```

```
for patient in db:
    print(patient.model_dump())
```

```
{'patient_id': 'P001', 'name': 'Rashed', 'age': 30, 'weight': 56.7, 'married': True, 'alle
{'patient_id': 'P002', 'name': 'Rashed', 'age': 40, 'weight': 70.0, 'married': True, 'alle
```

Why did not we use `list` and `dict` though? Because, we could make sure that the fields are list and string, but we could not check the data types inside those list or dict. That's why we used 2-step validation using `List[str]` and `Dict[str, str]`.

We could make our model more flexible. For example, not every patient will have allergies, but that field is required now! Let's work around that.