PYDANTIC - Python Library.

- We will use Pydantic with FastAPI, to define how the schema should be, in which the data is to be sent to the API endpoint.

- Pydantic is a complete, independent Python lib that can be used with any Python application. [It is not at all linked with FastAPI, in terms of functionality].

We are using it with FastAPI just to define the Schema.

[from pydantic import BaseModel
Import statement for using it.

- Now, we first need to think what field do we need, and what will be the datatype of the data in each field.

Ex:- Suppose, we are creating endpoint for creating posts by users. The field & values datatypes will be:

→ [title → str
  content → str

- So, we define the schema in Pydantic by making child class of the BaseModel class.
- In this class, we make fields (variables) that

we want to be the fields of data from user, and mention data type for each field.

For eg: We want to make schema class for the above thought fields. Then, the class will be like:-

```
class Post(BaseModel):
    title : str
    content : str
```

name of the class can be anything as wished.

the fields that we want in our schema.

datatypes of data in the fields

For each field, we need to mention datatypes [as we do in SQL schema definition], seperated by :

⇒ This Post class is called a Pydantic model. Now, we will use this Pydantic model as the datatype of the object that is received by the Path Operation function as parameter.

```
@app.post("/createposts")
def create_posts(new_post:Post):
    print(new_post)
    return {"data": "New Post created!"}
```

-> When using Pydantic model as type, extraction of data from fields is done automatically (without using Body(...))

- As now we are using the Pydantic model as the type of object in Parameter of Path Operation (the function actually), FastAPI will now automatically validate the data it receives from the client

Thus, it will validate whether data is provided for all the fields & whether they are of correct data-type. If not, then value error will be thrown.

Thus, the data obtained from user will now follow the schema strictly.

- In the received object, the keys (i.e, title & content here) will not be string anymore. They are variables (members of Pydantic Model class).
Thus, the data can be accessed like this :-

```
print(new_post.title)      # Pydantic Model type object obtained as parameter.
print(new_post.content)
```

Setting Default Value for a Field :-

- Like this, we can create a model, with a default valued field : If value for that field is not passed, then it will take that default value :-

1:17:30

```
class Post (BaseModel):
    title : str
    content : str
    published : bool = True
```

default value for the field

published → default valued field

Creating a Completely Optional Field :-

Completely optional field means users may or may not provide data for that field.
If no data is provided, that field will contain None.

```
from typing import Optional
---
class Post (BaseModel):
    rating : Optional[int] = None
    ---
    --
```

if data is provided, then data type of this field's value.

Thus, by using Pydantic library, we can ensure that we get the data exactly as we want.

1:20:45

dict() method :-

- dict() method belonging to the BaseModel class, converts and returns as a Pydantic model class object in the form of a dictionary.

Ex:- In the above Path operation :-

```
async def create_post(new_post: Post):
    ...
    post_dict = new_post.dict()
    print(type(post_dict))
    ...
    ...
```

o/p

```
<class 'dict'>
```

- The object that is returned by the method is a dictionary.
  The caller object remains a Pydantic model object.