**🔹 What is namedtuple?**

* namedtuple is a **special kind of tuple** from Python’s collections module.
* A normal tuple is accessed by **index** (t[0], t[1]), but a namedtuple lets you **give names to fields** so you can access them like object attributes (t.name, t.age).

👉 It’s like a **lightweight class** – no need to write a full class when you just need to store structured data.

**🔹 Importing**

from collections import namedtuple

**🔹 How to create a namedtuple**

# Define a namedtuple type

Person = namedtuple("Person", ["name", "age", "city"])

# Create objects

p1 = Person("Alice", 25, "Mumbai")

p2 = Person("Bob", 30, "Delhi")

print(p1) # Person(name='Alice', age=25, city='Mumbai')

print(p1.name) # Alice

print(p2.city) # Delhi

**🔹 Why use namedtuple?**

1. **Readable** – you can use person.name instead of person[0].
2. **Lightweight** – uses less memory than a class.
3. **Immutable** – values cannot be changed (just like normal tuples).
4. **Structured data** – perfect for configs, dataset entries, coordinates, etc.

**🔹 Example in DL Project**

You might use it to **store configurations or results**:

# Training results structure

TrainResult = namedtuple("TrainResult", ["epoch", "train\_loss", "val\_loss", "accuracy"])

result = TrainResult(epoch=5, train\_loss=0.32, val\_loss=0.28, accuracy=88.5)

print(result.accuracy) # 88.5

Or for **hyperparameters**:

HyperParams = namedtuple("HyperParams", ["lr", "batch\_size", "epochs"])

params = HyperParams(lr=0.001, batch\_size=32, epochs=20)

print(params.lr) # 0.001

**🔹 Summary**

* namedtuple = **tuple with named fields** (like a mini-class).
* Imported using from collections import namedtuple.
* Useful for:
  + Storing **configs** (lr, batch\_size, etc.).
  + Keeping **results** organized.
  + Making code **more readable and cleaner**.

👉 Do you want me to also show you a comparison of **when to use namedtuple vs dataclass** in DL projects (since dataclasses are often used nowadays too)?

Perfect 👍 this is an important distinction because **both namedtuple and dataclass are used in DL projects** for configs, results, etc. Let’s compare them clearly.

**🔹 namedtuple vs dataclass**

| **Feature** | **namedtuple** | **dataclass** |
| --- | --- | --- |
| **Import** | from collections import namedtuple | from dataclasses import dataclass |
| **Purpose** | Lightweight immutable objects with named fields | Lightweight classes with more flexibility |
| **Mutability** | Immutable (values cannot be changed once created) | Mutable by default (but can be frozen if needed) |
| **Field definition** | List of field names as strings | Use normal Python class attributes |
| **Readability** | Less readable (field names are strings) | Very readable, looks like a class |
| **Methods** | No custom methods (unless extended) | You can add methods, default values, type hints |
| **Memory usage** | Slightly lighter | Slightly heavier |
| **Use case in DL** | Store fixed configs or results (epoch, loss, acc) | Store configs, hyperparams, or objects that may change |

**🔹 Example 1: Using namedtuple**

from collections import namedtuple

# Define structure

TrainResult = namedtuple("TrainResult", ["epoch", "train\_loss", "val\_loss", "accuracy"])

# Create instance

res = TrainResult(epoch=5, train\_loss=0.32, val\_loss=0.28, accuracy=88.5)

print(res.accuracy) # 88.5

# res.accuracy = 90 # ❌ Error (immutable)

**🔹 Example 2: Using dataclass**

from dataclasses import dataclass

@dataclass

class TrainResult:

epoch: int

train\_loss: float

val\_loss: float

accuracy: float

# Create instance

res = TrainResult(epoch=5, train\_loss=0.32, val\_loss=0.28, accuracy=88.5)

print(res.accuracy) # 88.5

res.accuracy = 90 # ✅ Allowed (mutable)

**🔹 When to use which?**

✅ **Use namedtuple when:**

* You want something lightweight & immutable.
* You’re storing small, fixed pieces of data (like config snapshots, results per epoch).
* You don’t need methods, just structured data.

✅ **Use dataclass when:**

* You want more flexibility (mutability, default values, methods).
* You want type hints for better readability.
* Your data might evolve (e.g., add methods for computations, dynamic configs).
* Modern DL projects → **dataclasses are preferred** because of readability + features.

**✨ In short:**

* namedtuple = **immutable, lightweight, simple structured data**.
* dataclass = **modern, flexible, readable mini-class with more features**.

👉 Do you want me to also show you a **real DL project config example**, written once with namedtuple and once with dataclass, so you see the difference in practice?

Perfect 🙌 Let’s take a **realistic DL project config** (dataset info + training hyperparameters + model info) and see how it looks with both **namedtuple** and **dataclass**.

**🔹 Example DL Config with namedtuple**

from collections import namedtuple

# Define structures

DatasetConfig = namedtuple("DatasetConfig", ["train\_path", "val\_path", "batch\_size"])

TrainingConfig = namedtuple("TrainingConfig", ["epochs", "learning\_rate", "optimizer"])

ModelConfig = namedtuple("ModelConfig", ["type", "input\_size", "num\_classes"])

# Create configs

dataset\_cfg = DatasetConfig("data/train", "data/val", 32)

training\_cfg = TrainingConfig(epochs=20, learning\_rate=0.001, optimizer="adam")

model\_cfg = ModelConfig(type="ResNet50", input\_size=224, num\_classes=10)

# Usage

print(dataset\_cfg.batch\_size) # 32

print(training\_cfg.learning\_rate) # 0.001

print(model\_cfg.type) # ResNet50

# ❌ Cannot modify (immutable)

# training\_cfg.epochs = 30 # Error

👉 **Pros:** lightweight, fast, structured  
👉 **Cons:** immutable, harder to expand (no methods, no defaults)

**🔹 Example DL Config with dataclass**

from dataclasses import dataclass

@dataclass

class DatasetConfig:

train\_path: str

val\_path: str

batch\_size: int = 32 # Default value possible!

@dataclass

class TrainingConfig:

epochs: int

learning\_rate: float

optimizer: str = "adam" # Default value

@dataclass

class ModelConfig:

type: str

input\_size: int

num\_classes: int

# Create configs

dataset\_cfg = DatasetConfig("data/train", "data/val") # batch\_size defaults to 32

training\_cfg = TrainingConfig(epochs=20, learning\_rate=0.001)

model\_cfg = ModelConfig(type="ResNet50", input\_size=224, num\_classes=10)

# Usage

print(dataset\_cfg.batch\_size) # 32

print(training\_cfg.learning\_rate) # 0.001

print(model\_cfg.type) # ResNet50

# ✅ Can modify (mutable)

training\_cfg.epochs = 30

print(training\_cfg.epochs) # 30

👉 **Pros:** readable, flexible, supports defaults + type hints, can add methods  
👉 **Cons:** slightly heavier than namedtuple

**🔹 Where they fit in DL projects?**

* **namedtuple** → best for **logging results per epoch**, e.g. (epoch, loss, acc) snapshots.
* **dataclass** → best for **configs & hyperparameters**, because they may need defaults, mutability, and readability.

✅ **In short:**

* Use **namedtuple** when you need quick, immutable structures (like logging results).
* Use **dataclass** for configs, hyperparams, and structured project settings (preferred in modern DL projects).

👉 Do you want me to also show you how this **dataclass config** can be combined with a config.yaml file — so you load YAML into a dataclass automatically? That’s how many production DL projects are structured.

Great 🚀 Let’s connect **config.yaml** with **dataclass**. This way, your DL project stays clean, and configs are easy to change without touching Python code.

**🔹 Step 1: Create a config.yaml**

Example:

dataset:

train\_path: "data/train"

val\_path: "data/val"

batch\_size: 32

training:

epochs: 20

learning\_rate: 0.001

optimizer: "adam"

model:

type: "ResNet50"

input\_size: 224

num\_classes: 10

**🔹 Step 2: Define dataclass for configs**

from dataclasses import dataclass

import yaml

@dataclass

class DatasetConfig:

train\_path: str

val\_path: str

batch\_size: int = 32

@dataclass

class TrainingConfig:

epochs: int

learning\_rate: float

optimizer: str = "adam"

@dataclass

class ModelConfig:

type: str

input\_size: int

num\_classes: int

@dataclass

class Config:

dataset: DatasetConfig

training: TrainingConfig

model: ModelConfig

**🔹 Step 3: Load YAML into Dataclass**

def load\_config(path="config.yaml") -> Config:

with open(path, "r") as f:

config\_dict = yaml.safe\_load(f)

# Convert dict → dataclass

dataset\_cfg = DatasetConfig(\*\*config\_dict["dataset"])

training\_cfg = TrainingConfig(\*\*config\_dict["training"])

model\_cfg = ModelConfig(\*\*config\_dict["model"])

return Config(dataset=dataset\_cfg, training=training\_cfg, model=model\_cfg)

# Usage

cfg = load\_config()

print(cfg.dataset.batch\_size) # 32

print(cfg.training.learning\_rate) # 0.001

print(cfg.model.type) # ResNet50

**🔹 Why is this powerful?**

✅ Keeps **configs separate** from code → easier to experiment (just change YAML, no code edit).  
✅ **Dataclass ensures structure & type safety** → avoids silly mistakes like writing "thirty" instead of 30.  
✅ Easy to **log configs in training logs** → you know exactly what settings trained a model.  
✅ Scalable → add new sections in YAML, just update dataclass.

⚡ In most modern DL projects:

* **YAML** = storage of configs
* **dataclass** = structured way to use configs in Python
* Together → clean, maintainable, production-ready setup

👉 Do you want me to also show you how to **integrate Hydra/OmegaConf** so that YAML configs directly map into dataclasses (no manual \*\*kwargs unpacking)? That’s even more powerful and widely used in DL research.

Perfect 👍 Let’s now step into **Hydra/OmegaConf**, which are very popular in **research DL projects** (used in PyTorch Lightning, HuggingFace examples, etc).

Instead of writing code to manually parse YAML → dataclass, Hydra/OmegaConf can do it for you automatically.

**🔹 Step 1: Install Hydra & OmegaConf**

pip install hydra-core omegaconf

**🔹 Step 2: Write config.yaml**

Same as before:

dataset:

train\_path: "data/train"

val\_path: "data/val"

batch\_size: 32

training:

epochs: 20

learning\_rate: 0.001

optimizer: "adam"

model:

type: "ResNet50"

input\_size: 224

num\_classes: 10

**🔹 Step 3: Define dataclass configs**

Hydra/OmegaConf can **directly map YAML → dataclass**.

from dataclasses import dataclass

from omegaconf import OmegaConf

@dataclass

class DatasetConfig:

train\_path: str

val\_path: str

batch\_size: int = 32

@dataclass

class TrainingConfig:

epochs: int

learning\_rate: float

optimizer: str = "adam"

@dataclass

class ModelConfig:

type: str

input\_size: int

num\_classes: int

@dataclass

class Config:

dataset: DatasetConfig

training: TrainingConfig

model: ModelConfig

**🔹 Step 4: Load YAML → Dataclass (OmegaConf)**

# Load config.yaml into dataclass directly

cfg = OmegaConf.structured(Config) # Create schema

yaml\_cfg = OmegaConf.load("config.yaml") # Load YAML

cfg = OmegaConf.merge(cfg, yaml\_cfg) # Merge YAML into dataclass

# Access like normal attributes

print(cfg.dataset.batch\_size) # 32

print(cfg.training.learning\_rate) # 0.001

print(cfg.model.type) # ResNet50

**🔹 Step 5: Using Hydra for CLI-based overrides**

Hydra allows **changing configs from command line** without editing YAML.

import hydra

from omegaconf import DictConfig

@hydra.main(config\_path=".", config\_name="config", version\_base=None)

def main(cfg: DictConfig):

print(cfg.training.epochs) # from YAML

print(cfg.model.type) # from YAML

if \_\_name\_\_ == "\_\_main\_\_":

main()

Now, you can **override configs directly when running**:

python train.py training.epochs=50 model.type=EfficientNet

👉 It will use 50 epochs and model type EfficientNet without touching the YAML file.

**🔹 Why Hydra/OmegaConf are awesome in DL projects?**

✅ No manual parsing → YAML maps into structured configs automatically  
✅ Command-line overrides → experiment without editing files  
✅ Config composition → split configs (dataset.yaml, training.yaml, model.yaml) and merge them  
✅ Easy logging → configs are stored with training logs (reproducibility ✅)

⚡ Example: Many **PyTorch Lightning + Hydra** projects structure configs like this:

configs/

dataset.yaml

training.yaml

model.yaml

train.py

Then combine them automatically using Hydra.

👉 Do you want me to show you a **full DL training script** (with dataclasses + Hydra + config.yaml) so you see how it ties into model training?