Python Data Structures, OOP, Functions and PyTorch Data Pipelines

Your Name

Course / Lab

January 21, 2026

Roadmap

1. Core Python containers: lists, tuples, dictionaries
2. Tensors (concept + PyTorch intro later)
3. Classes (OOP): types of classes, constructors, inheritance, args/kwargs
4. Functions: kinds of functions, signatures, closures, decorators
5. Putting it together: classes that use functions
6. PyTorch basics: tensors, autograd, modules
7. DataLoader: batches, iteration, custom image dataset, MNIST

Why containers matter in ML code

- Your code moves data around constantly:
  - raw samples (paths, labels) in Python containers
  - batches in tensors
  - metrics and configs in dictionaries
- The big skill: choose the right container for the job.

Lists

- Ordered, mutable sequence: you can change it in place
- Great for: collections you append to, queues, building datasets, logs

List creation and basic operations

```
# Create lists
a = [1, 2, 3]
b = list((10, 20, 30))  # from a tuple

# Indexing and slicing
print(a[0])      # 1
print(a[-1])     # 3
print(a[1:3])    # [2, 3]

# Mutating
a.append(4)           # [1,2,3,4]
a.extend([5, 6])      # [1,2,3,4,5,6]
a[0] = 99             # [99,2,3,4,5,6]
```

List patterns you will use constantly

```
# 1) List comprehensions
squares = [x*x for x in range(5)]           # [0,1,4,9,16]
evens = [x for x in range(10) if x % 2==0] # [0,2,4,6,8]

# 2) Zip for paired iteration
xs = [1, 2, 3]
ys = [10, 20, 30]
pairs = list(zip(xs, ys))  # [(1,10),(2,20),(3,30)]

# 3) Enumerate for (index, value)
for i, v in enumerate(["cat", "dog"]):
    print(i, v)
```

Pitfalls: aliasing and copying

```
a = [1, 2, 3]
b = a           # alias: b points to same list
b.append(4)
print(a)        # [1,2,3,4]   surprise!

# copy (shallow)
c = a.copy()
c.append(5)
print(a)        # [1,2,3,4]
print(c)        # [1,2,3,4,5]
```

- For nested lists, shallow copy copies the outer list only.

Tuples

- Ordered, immutable sequence: cannot change elements in place
- Great for: fixed-size records, returning multiple values, keys in dicts
- ML habit: represent shapes as tuples, e.g. `(N, C, H, W)`

Tuple examples

```
# Create tuples
t = (128, 3, 32, 32)    # shape
one = (5,)              # single-element tuple needs a comma

# Unpacking
x, y = (10, 20)
print(x, y)

# Returning multiple values
def minmax(xs):
    return min(xs), max(xs)

lo, hi = minmax([3, 1, 9])
print(lo, hi)  # 1 9
```

Dictionaries

- Key-value mapping: fast lookup by key
- Great for:
  - configuration (hyperparameters)
  - a sample described by fields: `{"image": ..., "label": ...}`
  - logging metrics: `{"loss": ..., "acc": ...}`

Dictionary basics

```
cfg = {
    "batch_size": 64,
    "lr": 1e-3,
    "model": "mlp",
}

print(cfg["lr"])                # 0.001
cfg["epochs"] = 10              # add
cfg["lr"] = 5e-4                # update

# Safe get with default
wd = cfg.get("weight_decay", 0.0)

# Iterate
for k, v in cfg.items():
    print(k, v)
```

Dictionary patterns: counting and grouping

```python
# Counting
counts = {}
for x in ["a", "b", "a", "c", "a"]:
    counts[x] = counts.get(x, 0) + 1
print(counts)  # {'a':3, 'b':1, 'c':1}

# Grouping values by key
groups = {}
pairs = [("cat", 1), ("dog", 2), ("cat", 3)]
for k, v in pairs:
    groups.setdefault(k, []).append(v)
print(groups)  # {'cat':[1,3], 'dog':[2]}
```

Tensors (conceptual)

- A tensor is an N-dimensional array:
  - scalar: 0D
  - vector: 1D
  - matrix: 2D
  - image batch: 4D (N, C, H, W)
- In ML, tensors hold your data and your model parameters.
- Python lists can represent arrays, but tensors are faster and support gradients.

Classes: what and why

- A class is a blueprint for objects
- An object bundles:
  - **state** (attributes, stored on `self`)
  - **behavior** (methods)
- ML connection: a neural net is a class with parameters (state) and a forward pass (behavior)

A minimal class

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def norm2(self):
        return self.x**2 + self.y**2

p = Point(3, 4)
print(p.x, p.y)       # 3 4
print(p.norm2())      # 25
```

Constructors and attributes

- `__init__` is the constructor-like initializer
- Typical pattern:
  - validate inputs
  - store configuration on `self`
  - create any internal state
- Objects can also have attributes added later, but in ML code prefer explicit attributes.

Instance methods vs class methods vs static methods

```python
class Temperature:
    scale = "C"  # class attribute, shared by all instances

    def __init__(self, celsius):
        self.celsius = float(celsius)

    def to_f(self):  # instance method
        return self.celsius * 9/5 + 32

    @classmethod
    def from_f(cls, f):  # class method: alternative constructor
        c = (f - 32) * 5/9
        return cls(c)

    @staticmethod
    def is_valid(c):  # static method: utility, no self/cls needed
        return c > -273.15
```

Types of classes you will see

- **Data classes / record-like**: mostly store values (config, sample)
- **Service classes**: manage resources (logger, checkpoint manager)
- **Model classes**: have parameters and computation (PyTorch modules)
- **Factory classes**: create other objects (dataset builders)

Record-like class with defaults

```
class TrainConfig:
    def __init__(self, lr=1e-3, batch_size=64, epochs=10):
        self.lr = lr
        self.batch_size = batch_size
        self.epochs = epochs

cfg = TrainConfig(epochs=20)
print(cfg.lr, cfg.batch_size, cfg.epochs)
```

*args and **kwargs (the Swiss Army funnel)

- `*args`: extra positional arguments packed into a tuple
- `**kwargs`: extra keyword arguments packed into a dictionary
- Useful when:
  - forwarding options to another function/class
  - writing flexible APIs (layers, transforms, loggers)

*args and **kwargs examples

```
def show(*args, **kwargs):
    print("args:", args)        # tuple
    print("kwargs:", kwargs)    # dict

show(1, 2, 3, name="Sayan", debug=True)

# Unpacking when calling
xs = (10, 20)
opts = {"sep": " | ", "end": "!\n"}
print(*xs)              # print(10, 20)
print("hello", **opts)
```

Forwarding kwargs in classes (common in ML)

```
class Logger:
    def __init__(self, prefix="train"):
        self.prefix = prefix

    def log(self, **metrics):
        # metrics is a dict
        print(self.prefix, metrics)

class Trainer:
    def __init__(self, logger):
        self.logger = logger

    def step(self, loss, acc):
        self.logger.log(loss=loss, acc=acc)

trainer = Trainer(Logger(prefix="exp1"))
trainer.step(loss=0.25, acc=0.91)
```

Inheritance and hierarchy

- Inheritance lets a class reuse and extend behavior
- Parent class defines shared interface
- Child class overrides or adds methods
- In ML: base `Dataset`, base `nn.Module`, custom specializations

Inheritance example: base + specialized

```
class BaseMetric:
    def update(self, y_pred, y_true):
        raise NotImplementedError

    def compute(self):
        raise NotImplementedError

class Accuracy(BaseMetric):
    def __init__(self):
        self.correct = 0
        self.total = 0

    def update(self, y_pred, y_true):
        self.correct += sum(int(p==t) for p, t in zip(y_pred, y_true
            ))
        self.total += len(y_true)

    def compute(self):
```

Composition vs inheritance

```python
# Composition: "has-a"
class ModelWithPreprocess:
    def __init__(self, model, preprocess_fn):
        self.model = model
        self.preprocess_fn = preprocess_fn

    def __call__(self, x):
        x = self.preprocess_fn(x)
        return self.model(x)
```

- Composition is often cleaner than deep inheritance trees.

Useful dunder methods in ML code

```
class Foo:
    def __repr__(self):
        return "Foo()"

    def __call__(self, x):
        return x * 2

f = Foo()
print(f)        # Foo()
print(f(10))    # 20
```

- `__call__` makes instances behave like functions (common for transforms).

Functions: the building blocks

- A function maps inputs to outputs
- In ML pipelines: transforms, losses, metrics, schedulers
- Good functions:
  - have clear signatures
  - avoid hidden state (unless intentional)
  - are easy to test on small inputs

Kinds of functions you will meet

```python
# 1) Regular named function
def add(a, b):
    return a + b

# 2) Lambda (small anonymous function)
square = lambda x: x*x

# 3) Generator function (yields values lazily)
def batchify(xs, batch_size):
    for i in range(0, len(xs), batch_size):
        yield xs[i:i+batch_size]
```

Default args, keyword-only args

```python
def clip(x, lo=0.0, hi=1.0):
    return min(max(x, lo), hi)

def f(a, b, *, debug=False):  # debug must be passed by name
    if debug:
        print("debug:", a, b)
    return a + b

print(clip(1.5))          # 1.0
print(f(2, 3, debug=True))
```

Closures: functions that remember

```
def make_scaler(alpha):
    def scale(x):
        return alpha * x
    return scale

times3 = make_scaler(3)
print(times3(10))  # 30
```

- Closures are handy for building simple parameterized transforms.

Decorators: wrapping functions

```python
import time

def timing(fn):
    def wrapped(*args, **kwargs):
        t0 = time.time()
        out = fn(*args, **kwargs)
        t1 = time.time()
        print(fn.__name__, "took", round(t1 - t0, 4), "s")
        return out
    return wrapped

@timing
def slow_sum(n):
    return sum(range(n))

slow_sum(1_000_000)
```

Classes with functions: a common ML pattern

- A class often:
  - stores configuration
  - exposes methods that call helper functions
- This keeps the code modular and readable.

Example: normalizer class using helper functions

```
def mean_std(xs, eps=1e-8):
    mu = sum(xs) / len(xs)
    var = sum((x - mu)**2 for x in xs) / len(xs)
    return mu, (var + eps)**0.5

class Normalizer:
    def __init__(self, eps=1e-8):
        self.eps = eps
        self.mu = None
        self.std = None

    def fit(self, xs):
        self.mu, self.std = mean_std(xs, eps=self.eps)
        return self

    def transform(self, xs):
        if self.mu is None:
            raise ValueError("Call fit() first.")
```

PyTorch basics: what to learn first

- Tensors: shapes, dtypes, devices
- Autograd: gradients and `backward()`
- Modules: building models as classes
- Optimizers: updating parameters
- Data pipeline: datasets, dataloaders, batches

PyTorch tensors: creation and basic ops

```python
import torch

x = torch.randn(4, 3)            # shape [4,3]
y = torch.zeros(4, 3)
z = x + y                        # elementwise
m = x @ torch.randn(3, 2)        # matmul -> [4,2]

print(x.shape, x.dtype, x.device)

# Reshape
a = torch.randn(2, 3, 4)
b = a.view(2, 12)                # same storage, different view
c = a.reshape(2, 12)             # safe reshape
```

Device: CPU vs GPU

```
device = "cuda" if torch.cuda.is_available() else "cpu"

x = torch.randn(32, 128).to(device)
w = torch.randn(128, 64).to(device)
y = x @ w

print(y.device)
```

Autograd: gradients

```
import torch

w = torch.randn(3, requires_grad=True)
x = torch.tensor([1.0, 2.0, 3.0])

loss = (w * x).sum()
loss.backward()

print(w.grad)  # gradient of loss wrt w is x
```

Models are classes: `nn.Module`

- In PyTorch, you define a model by subclassing `nn.Module`
- `__init__`: define layers
- `forward`: define computation
- Parameters are discovered via `model.parameters()`

A simple model (MLP)

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self, in_dim=784, hidden=256, out_dim=10, dropout
        =0.0):
        super().__init__()
        self.fc1 = nn.Linear(in_dim, hidden)
        self.fc2 = nn.Linear(hidden, out_dim)
        self.drop = nn.Dropout(dropout)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.drop(x)
        return self.fc2(x)

model = MLP(dropout=0.1)
```

Loss + optimizer + a minimal train step

```
loss_fn = nn.CrossEntropyLoss()
opt = torch.optim.Adam(model.parameters(), lr=1e-3)

x = torch.randn(64, 784)           # batch of features
y = torch.randint(0, 10, (64,))    # batch of labels

model.train()
opt.zero_grad()
logits = model(x)
loss = loss_fn(logits, y)
loss.backward()
opt.step()
```

Dataset and DataLoader: the data conveyor belt

- `Dataset`: defines how to get one sample (`__getitem__`)
- `DataLoader`: creates batches, shuffles, parallelizes loading
- Batches usually look like:
  - `x`: tensor $(N, ...)$
  - `y`: tensor $(N,)$ or $(N, ...)$

A toy Dataset and DataLoader

```python
import torch
from torch.utils.data import Dataset, DataLoader

class ToyDataset(Dataset):
    def __init__(self, n=1000):
        self.x = torch.randn(n, 10)
        self.y = (self.x.sum(dim=1) > 0).long()  # 0/1 labels

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

ds = ToyDataset(n=500)
loader = DataLoader(ds, batch_size=64, shuffle=True)

xb, yb = next(iter(loader))
```

Iterating over batches

```
for step, (x, y) in enumerate(loader):
    # x: [batch, features]
    # y: [batch]
    if step == 0:
        print("first batch shapes:", x.shape, y.shape)
    # training code would go here
```

Common DataLoader knobs

- `batch_size`: how many samples per batch
- `shuffle`: randomize order each epoch (usually True for training)
- `num_workers`: parallel loading processes
- `pin_memory`: speed transfer to GPU
- `drop_last`: drop the last incomplete batch

Training loop with batches (complete skeleton)

```
device = "cuda" if torch.cuda.is_available() else "cpu"
model = MLP(in_dim=10, hidden=64, out_dim=2).to(device)
loss_fn = nn.CrossEntropyLoss()
opt = torch.optim.Adam(model.parameters(), lr=1e-3)

for epoch in range(3):
    model.train()
    total_loss = 0.0

    for x, y in loader:
        x, y = x.to(device), y.to(device)

        opt.zero_grad()
        logits = model(x)
        loss = loss_fn(logits, y)
        loss.backward()
        opt.step()
```

Custom image dataset: two common layouts

- **Folder per class** (ImageFolder style)
  - `data/train/cat/*.jpg`
  - `data/train/dog/*.jpg`
- **CSV annotations**
  - `path,label` per line

Option A: use torchvision ImageFolder

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

tfm = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),  # [0,1], shape [C,H,W]
])

train_ds = datasets.ImageFolder("data/train", transform=tfm)
train_loader = DataLoader(train_ds, batch_size=32, shuffle=True,
   num_workers=4)

x, y = next(iter(train_loader))
print(x.shape, y.shape)  # [32,3,128,128] [32]
```

Option B: write your own custom image Dataset

```python
import os
from PIL import Image
from torch.utils.data import Dataset

class CustomImageDataset(Dataset):
    def __init__(self, items, transform=None):
        """
        items: list of tuples (path, label)
        transform: torchvision transform applied to PIL image
        """
        self.items = items
        self.transform = transform

    def __len__(self):
        return len(self.items)

    def __getitem__(self, idx):
```

Custom dataset + DataLoader

```python
from torchvision import transforms
from torch.utils.data import DataLoader

tfm = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
])

ds = CustomImageDataset(items, transform=tfm)
loader = DataLoader(ds, batch_size=2, shuffle=False)

x, y = next(iter(loader))
print(x.shape, y)  # [2,3,64,64] tensor([0,1])
```

Handling variable-size samples: `collate_fn`

- Some datasets have variable image sizes or variable-length sequences.
- Default collation stacks tensors, so shapes must match.
- Solution: write a `collate_fn` that pads or returns lists.

Example collate_fn that returns lists

```
def collate_as_list(batch):
    xs, ys = zip(*batch)      # tuples of length B
    return list(xs), torch.tensor(ys)

loader = DataLoader(ds, batch_size=8, collate_fn=collate_as_list)
xs, ys = next(iter(loader))
print(type(xs), len(xs), ys.shape)
```

MNIST: the hello-world dataset (digits)

- 28x28 grayscale images of digits 0-9
- Good for learning:
  - transforms
  - dataloaders
  - simple models and training loops

Load MNIST with torchvision

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

tfm = transforms.Compose([
    transforms.ToTensor(),                  # [1,28,28]
    transforms.Normalize((0.1307,), (0.3081,)),  # common MNIST
        stats
])

train_ds = datasets.MNIST(
    root="data", train=True, download=True, transform=tfm
)
test_ds = datasets.MNIST(
    root="data", train=False, download=True, transform=tfm
)

train_loader = DataLoader(train_ds, batch_size=64, shuffle=True,
```

A small MNIST model (MLP)

```python
import torch.nn as nn
import torch.nn.functional as F

class MNISTMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 256)
        self.fc2 = nn.Linear(256, 10)

    def forward(self, x):
        # x: [N,1,28,28]
        x = x.view(x.shape[0], -1)      # flatten -> [N,784]
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

Training MNIST: full loop with evaluation

```
import torch

device = "cuda" if torch.cuda.is_available() else "cpu"
model = MNISTMLP().to(device)
opt = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

def accuracy(model, loader):
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            logits = model(x)
            pred = logits.argmax(dim=1)
            correct += (pred == y).sum().item()
            total += y.numel()
    return correct / total
```

A small MNIST model (CNN version)

```
class MNISTCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)  # 28x28
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1) # 28x28
        self.pool = nn.MaxPool2d(2)                               # /2
        self.fc = nn.Linear(32 * 14 * 14, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(F.relu(self.conv2(x)))  # [N,32,14,14]
        x = x.view(x.shape[0], -1)
        return self.fc(x)
```

Mental checklist when coding ML in Python

- Containers: list/tuple/dict are for structure, tensors are for math
- Shapes: always know what (N, C, H, W) is at each step
- OOP: models and datasets are classes with clear interfaces
- DataLoader: batches are just samples stacked together
- Workflow: overfit a tiny subset, then scale up

Exercises (fast, useful)

1. Write a `Dataset` that returns `{"x": x, "y": y}` dicts.
2. Add a transform class with `__call__` to normalize an image tensor.
3. Modify MNIST model to print shapes in `forward` for the first batch only.

Questions?

Bring a bug, a tensor shape, or a suspicious loss curve.