# Module 1

**Tensors and Datasets**

**Dataset**

## 📌 Simple Dataset

### 🔹 Creating a Dataset Class

A dataset object is created by subclassing the abstract Dataset class provided by PyTorch.

Within the constructor:

* Input features and target values are stored as tensors (x and y), each containing 100 samples. The values are created in the object constructor and assigned to the **self.x** and **self.y** tensors
* The total number of samples is stored in a length attribute.

The dataset class overrides two core methods:

* **\_\_len\_\_**: Returns the number of samples.
* **\_\_getitem\_**\_: Accepts an index and returns a tuple of feature and target tensors corresponding to that index.

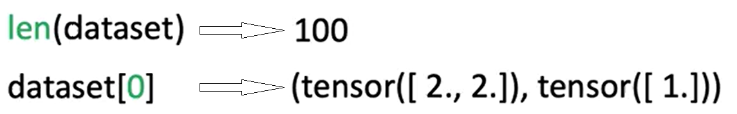
|  |  |
| --- | --- |
|  |  |

🔸 **Accessing Data Samples:**

Individual samples are retrieved using square brackets, which act as a proxy for the \_\_getitem\_\_ method.

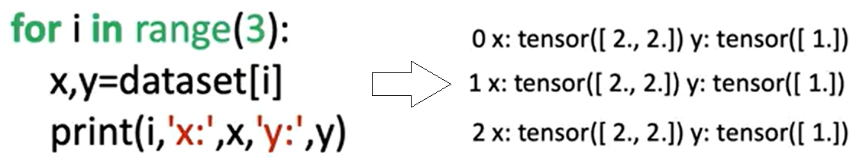
This method returns a tuple:

* The first element corresponds to a feature tensor.
* The second element corresponds to a target tensor.



The dataset behaves like an iterable. It can be accessed using index notation or through iteration in a loop:

* Iterating over the dataset triggers repeated calls to \_\_getitem\_\_, returning one sample per iteration.

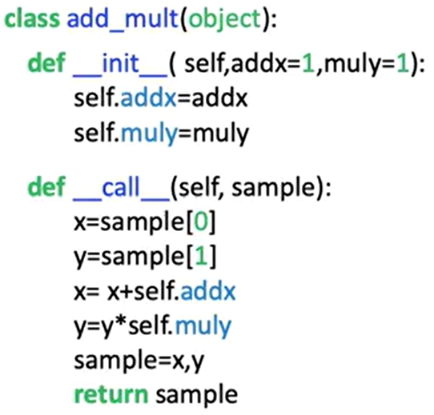


### 🔹 Applying Transforms to a Dataset

Transformations can be applied to samples using **callable classes** instead of standalone functions.

These classes define a **\_\_call\_\_** method, allowing them to behave like functions when passed to the dataset.

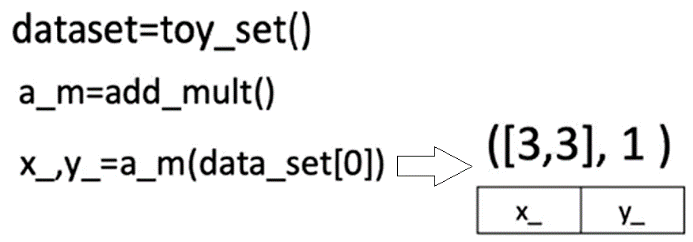
**Custom Transform Class**

* A custom transformation class is defined with two parameters:
  + One to add a constant to the feature tensor.
  + One to multiply the target tensor by a constant.
* When a sample is passed to this transformation object, the transformation is applied and the modified tensors are returned as a tuple.

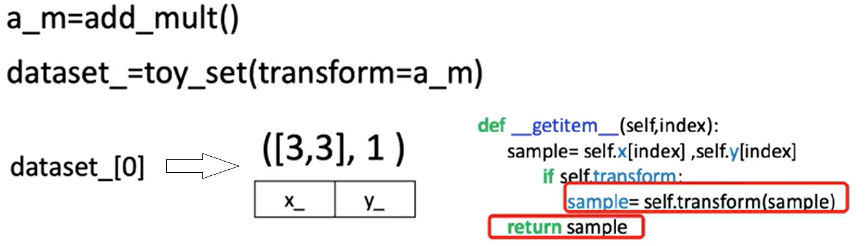
🔸 **Applying Transforms to Dataset Samples:**

There are two methods for applying a transformation:

1. **Manual Application**:
   * The transformation object is created separately.
   * The object is manually applied to a sample retrieved from the dataset.
   * Only the selected sample is transformed.



1. **Automatic Application via Constructor:**
   * The transformation object is passed to the dataset class during initialization.
   * Inside the dataset class, the transform parameter is assigned.
   * During each call to **\_\_getitem\_\_,** the transformation is applied automatically to every sample.
   * This ensures that the transformation is consistently applied across all retrieved data.



### 🔹 Composing Multiple Transforms

PyTorch provides a Compose class for chaining multiple transformations.

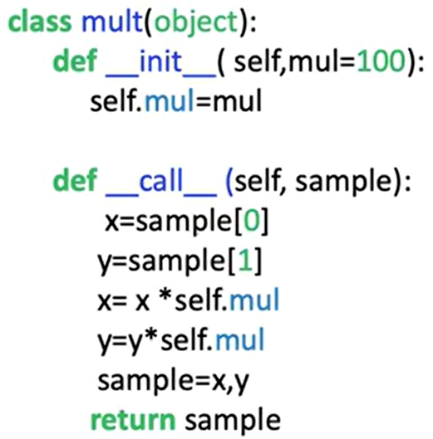
A list of transformation objects is passed to the Compose constructor.

When a sample is passed to the composed transform:

* The first transformation is applied.
* The output is passed to the second transformation.
* The final output is returned as a transformed tuple of tensors.

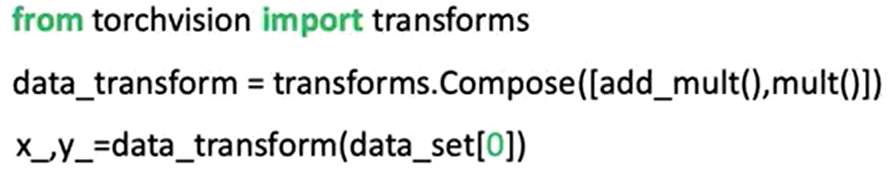
This compose object can be passed into the dataset class, enabling **automatic application of multiple transformations** during sample retrieval.

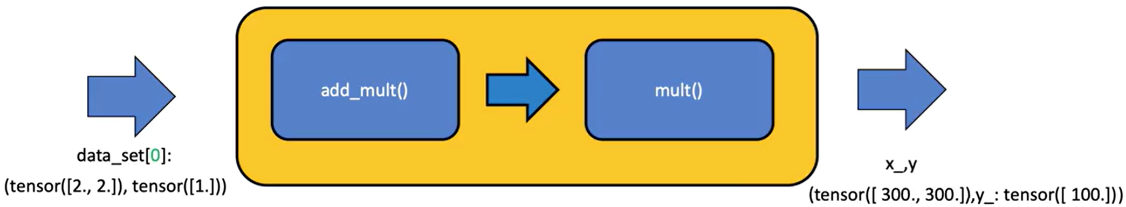
Let's say we would like to apply another transform, the class “**mult**” will multiply all the elements of a tensor by the value mul.



1. **Manual Application**:

In the constructor, we place a list. The first element of the list is the constructor for the first transform, the second element of the list is the constructor for the second transform.





1. **Automatic Application via Constructor:**

The compose object can be applied directly in the dataset constructor, each time a sample is retrieved, the original tensor is passed to the compose object (the first transform is applied, then the second transform is applied).



### ✅ Takeaways

✅Custom dataset objects can be built by subclassing PyTorch’s Dataset class and implementing the length and indexing methods.

✅Data stored in tensors can be accessed, indexed, and iterated over in a structured and repeatable way.

✅Transformations can be implemented as callable classes for better modularity and reuse.

✅Applying transformations during dataset construction enables efficient preprocessing at the data loading stage.

✅Multiple transformations can be composed using PyTorch's Compose utility, allowing sequential data processing in a clean and scalable manner.