Hands On Programming Session

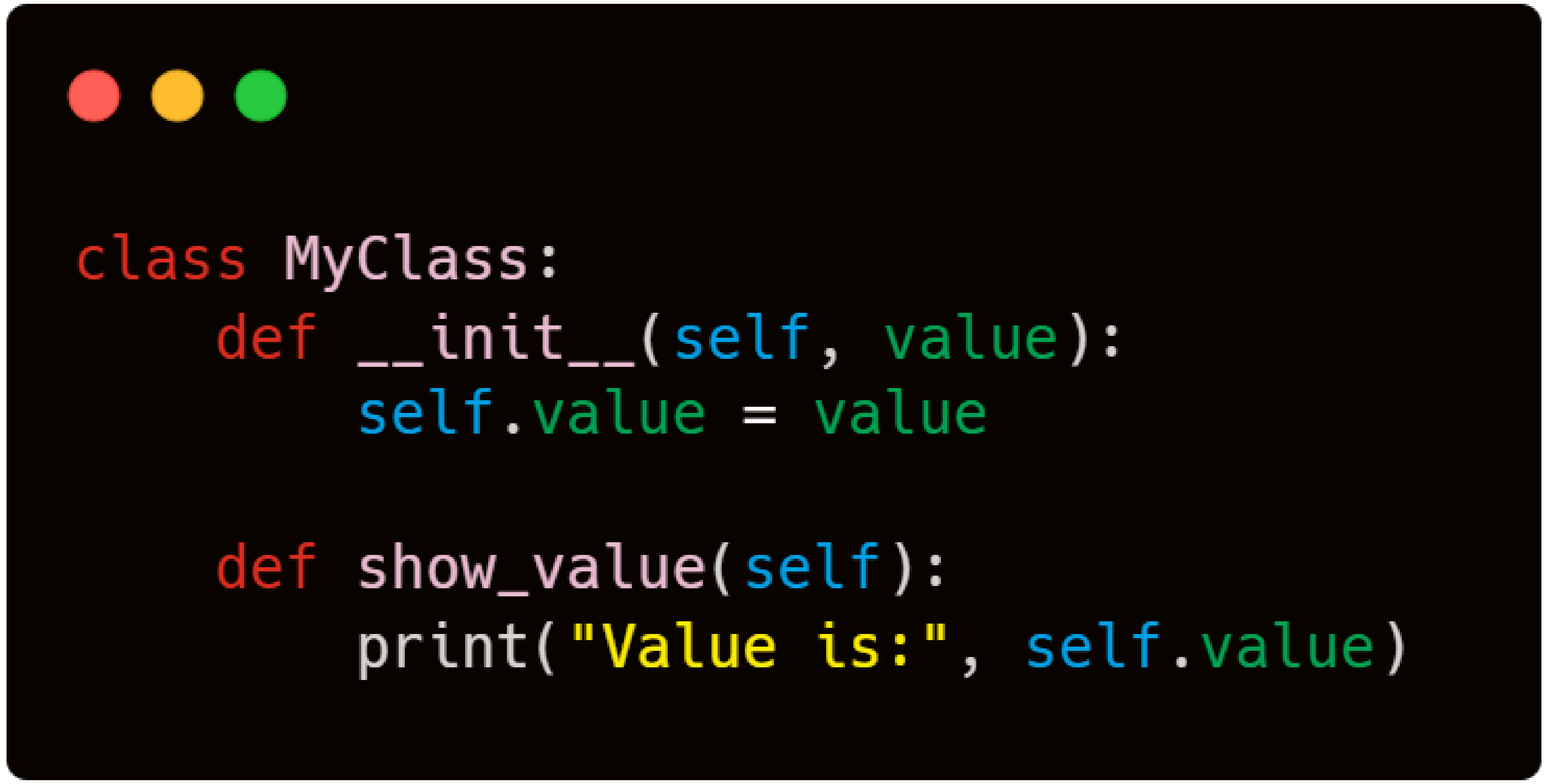
- 1/22/2025

Brighter World

McMaster University

Some Python Concepts

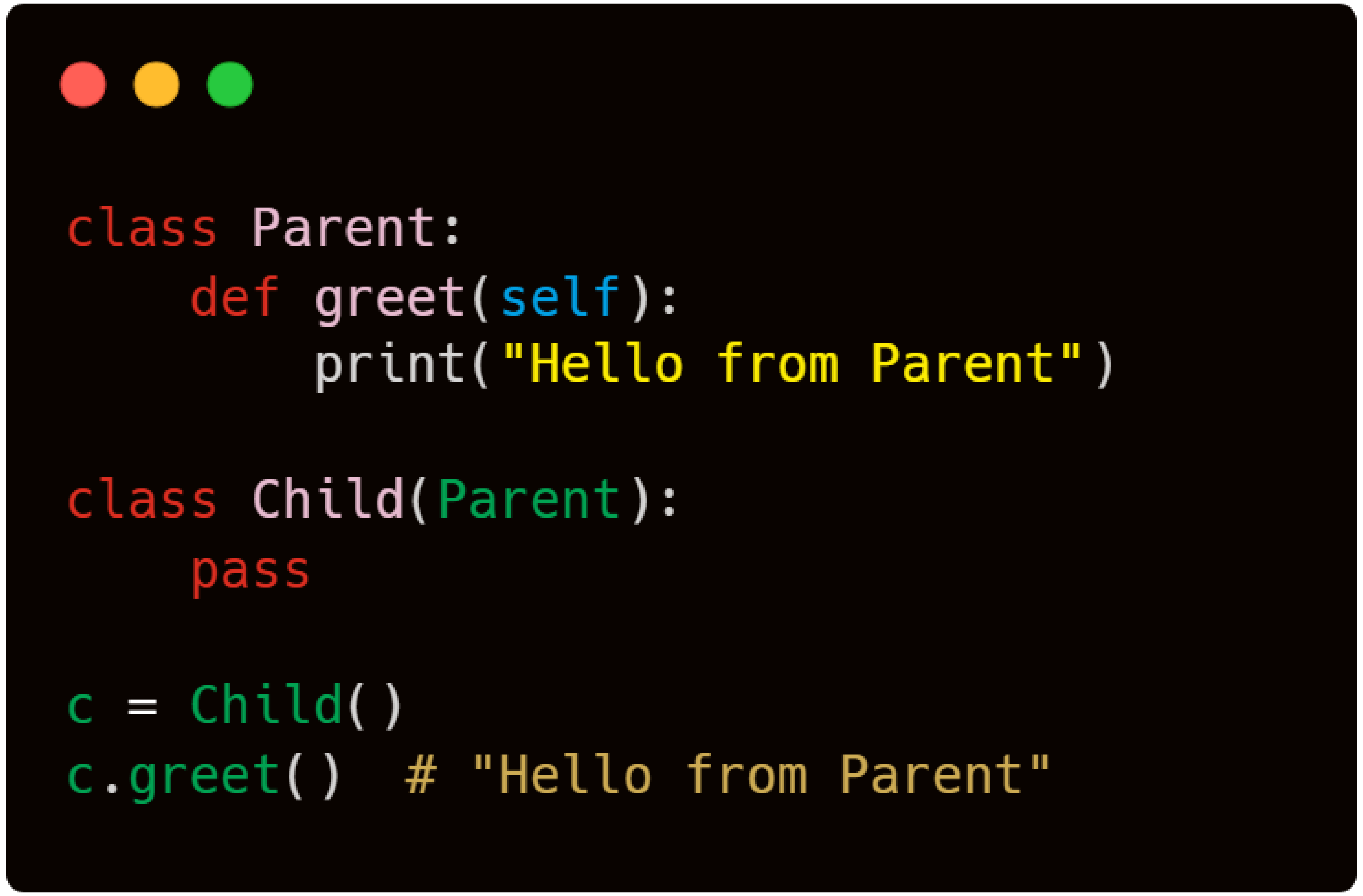
Classes



```
class MyClass:
    def __init__(self, value):
        self.value = value

    def show_value(self):
        print("Value is:", self.value)
```

- **Class:** A blueprint for creating objects, combining data (attributes) and behaviors (methods) for reusable and organized programming.
- **Constructor:** A special method (__init__) called automatically during object creation to initialize attributes.
- **Instance Variable:** A variable tied to an object instance, unique to each created object of the class.

Inheritance



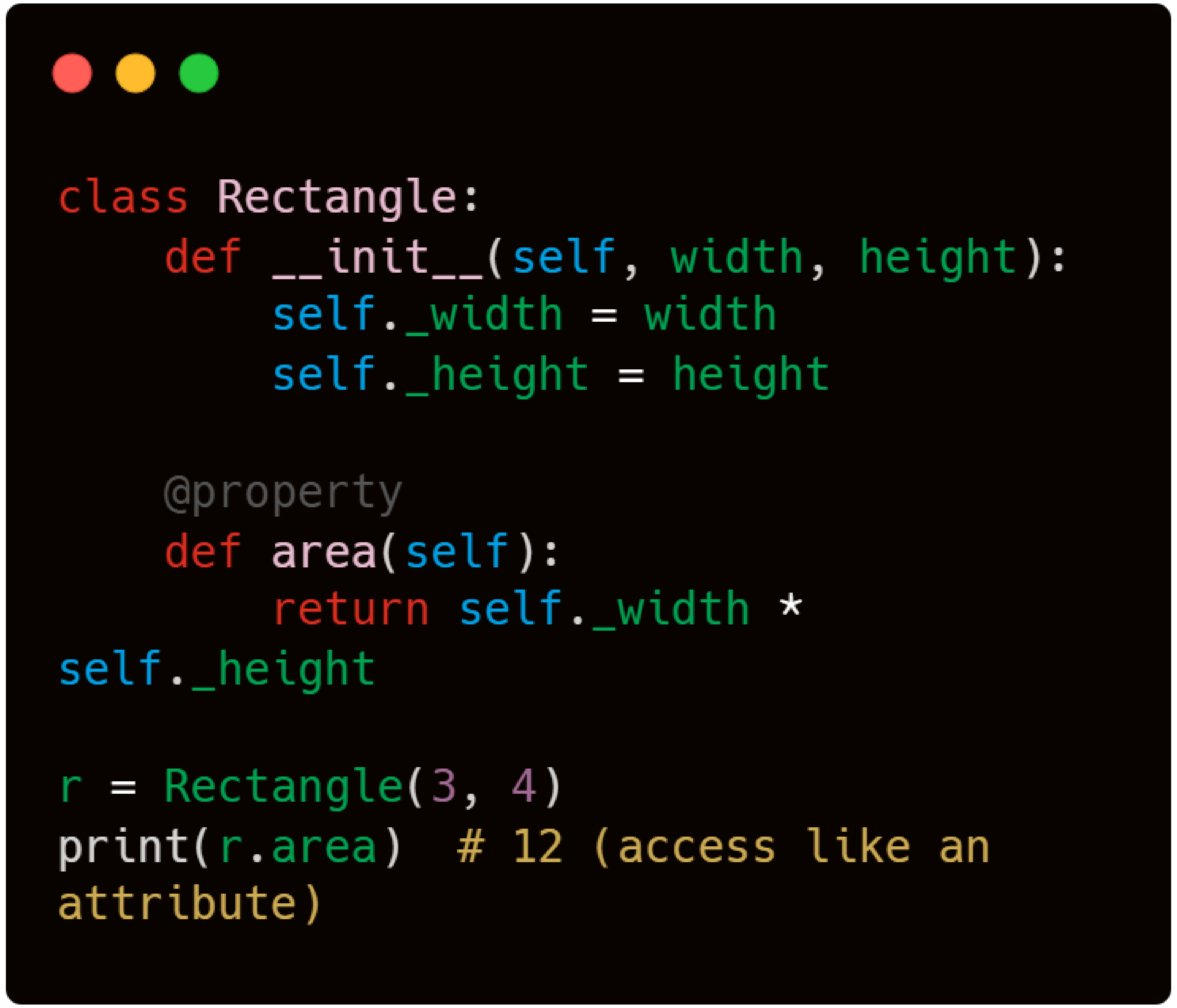
```python
class Parent:
    def greet(self):
        print("Hello from Parent")

class Child(Parent):
    pass

c = Child()
c.greet()  # "Hello from Parent"
```

- **Inheritance:** A mechanism that allows a class (child) to inherit attributes and methods from another class (parent), promoting code reuse and hierarchy.

- **Parent Class:** The class whose properties and methods are inherited by another class.

- **Child Class:** The class that inherits from the parent, gaining its functionality while allowing for extensions or overrides.

Decorators



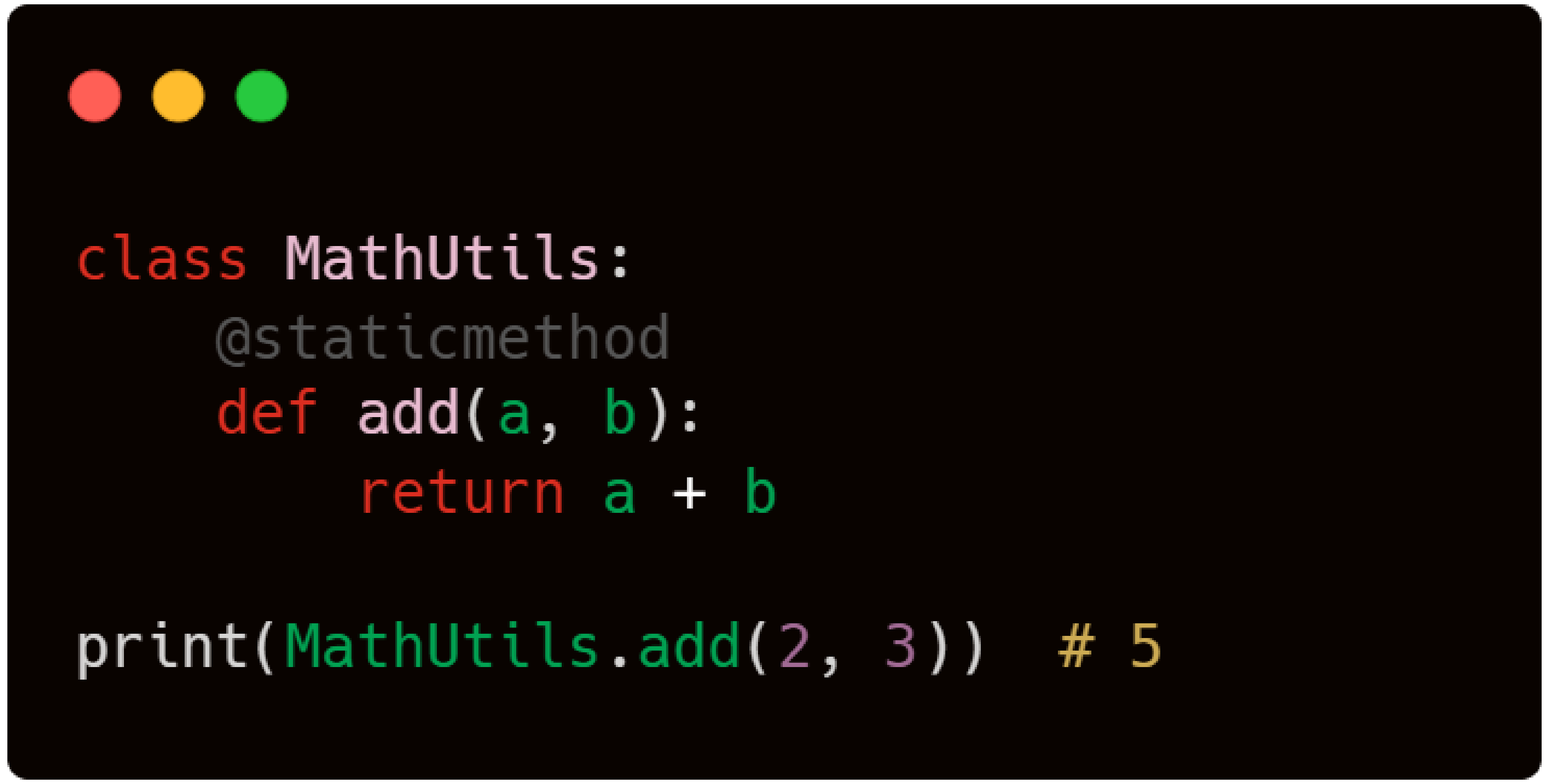
```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    @property
    def area(self):
        return self._width *
self._height

r = Rectangle(3, 4)
print(r.area)  # 12 (access like an
attribute)
```

- **Decorators:** Special functions or symbols (@) in Python that modify or enhance the behavior of functions, methods, or classes without changing their source code.

@property:

- Transforms a method into a read-only attribute, allowing access without explicit method calls.

Decorators



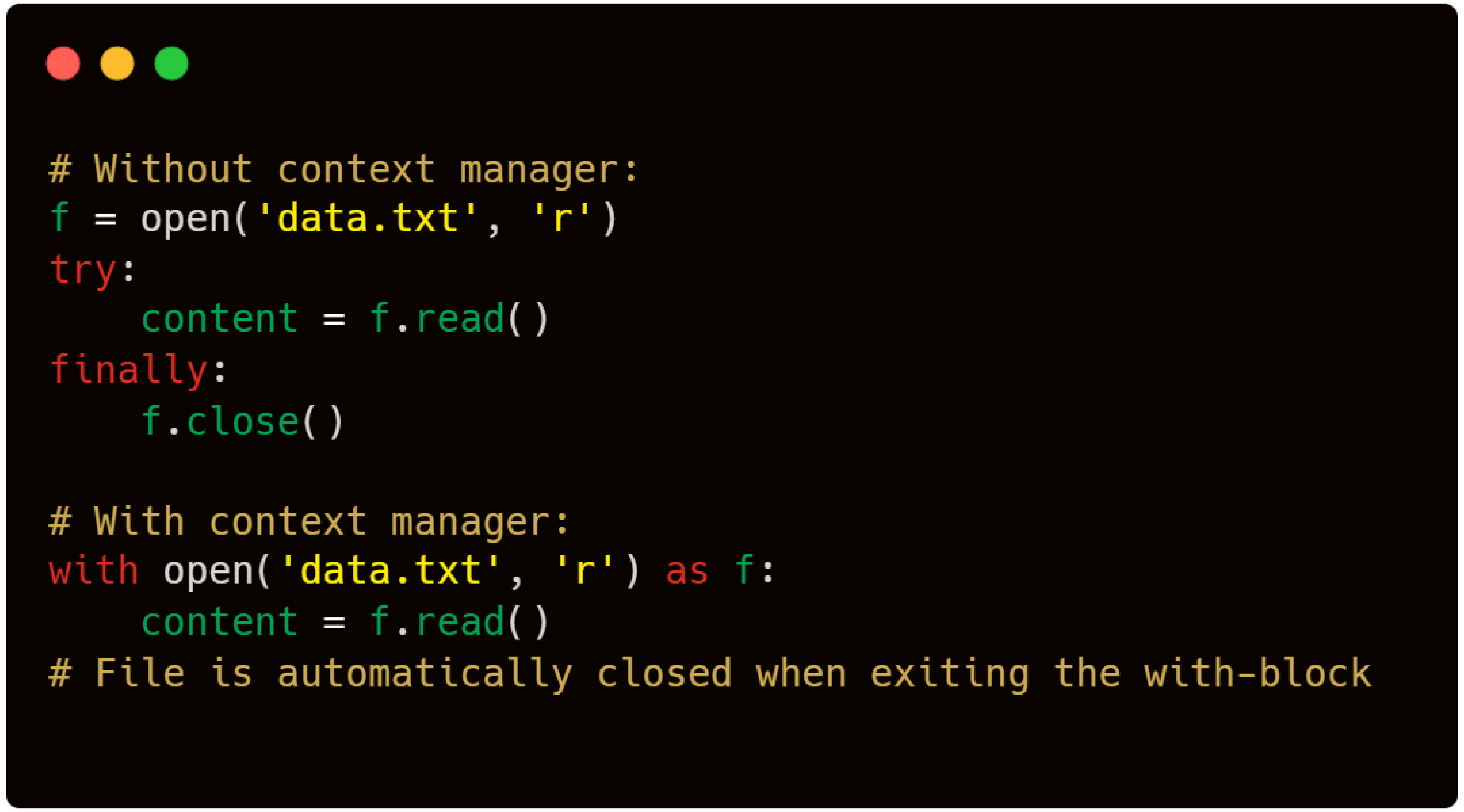
```python
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

print(MathUtils.add(2, 3))  # 5
```

- **Decorators:** Special functions or symbols (@) in Python that modify or enhance the behavior of functions, methods, or classes without changing their source code.

- **@staticmethod:**

Defines a method that belongs to the class rather than an instance, and it doesn't access or modify class/instance-level attributes.

Context Managers



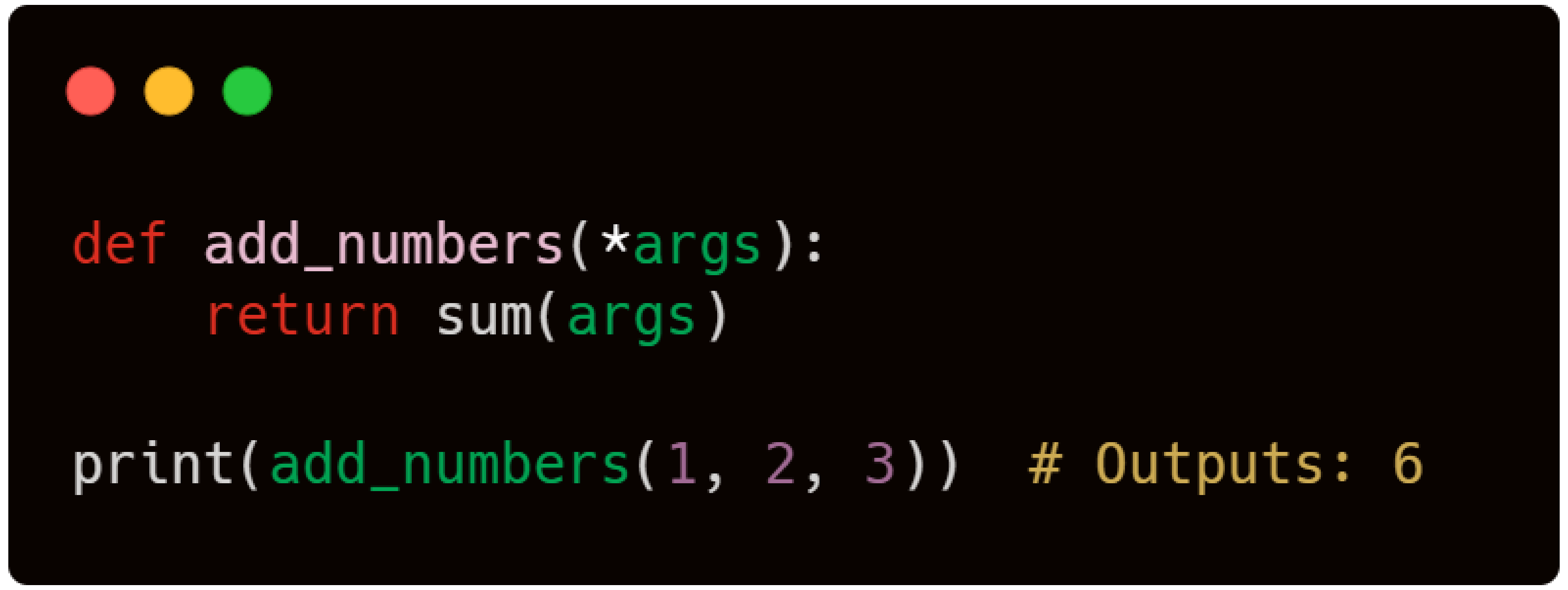
```python
# Without context manager:
f = open('data.txt', 'r')
try:
    content = f.read()
finally:
    f.close()

# With context manager:
with open('data.txt', 'r') as f:
    content = f.read()
# File is automatically closed when exiting the with-block
```

- **Context Managers:** A Python construct that manages resources when entering and exiting a block of code.

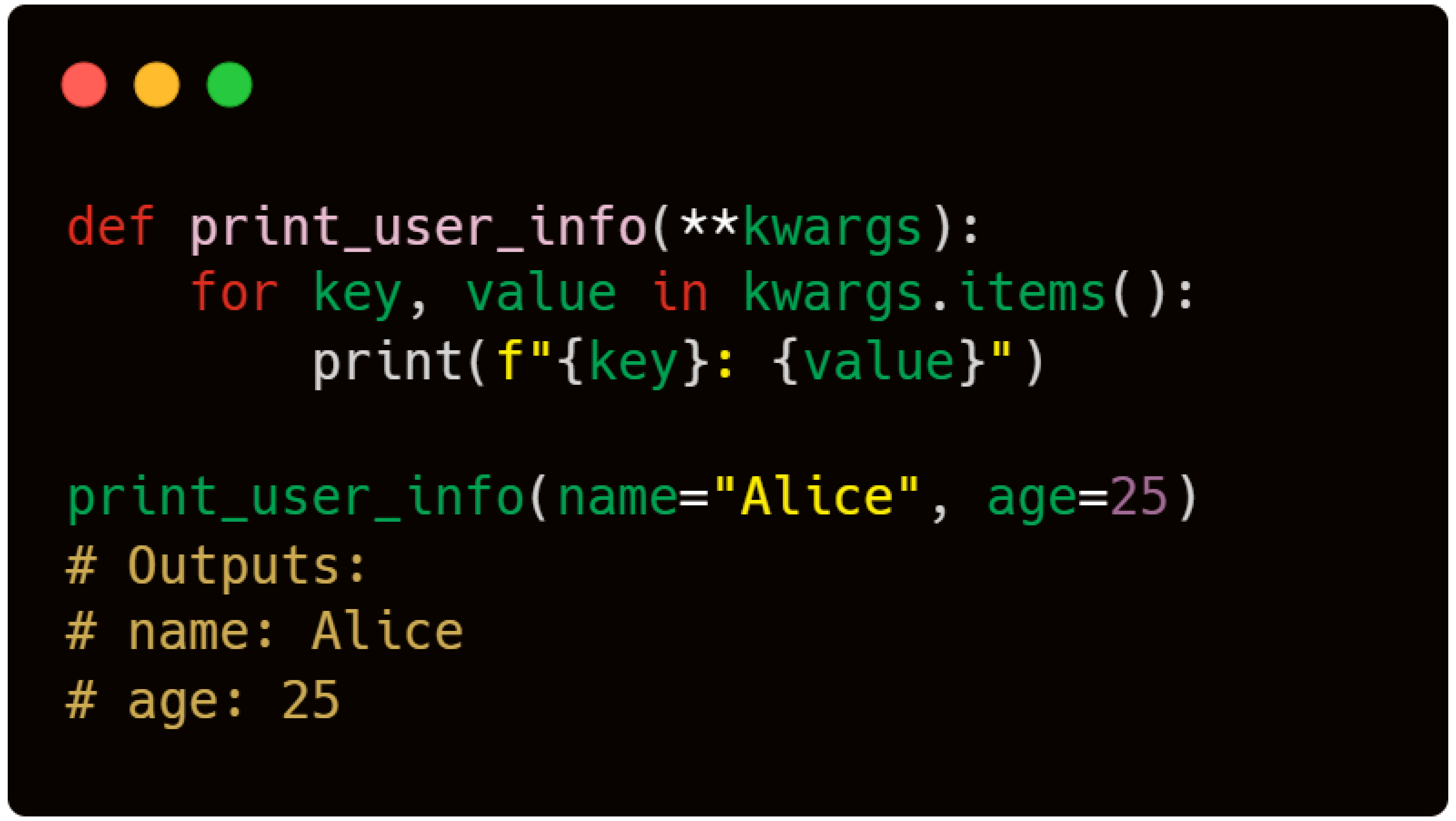
Why we use it?

- Automatically handles opening/closing resources (like files or network connections) even if exceptions occur.
- Keeps resource-management code clean and less error-prone.

Positional & Keyword Arguments



```python
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3))  # Outputs: 6
```



```python
def print_user_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_user_info(name="Alice", age=25)
# Outputs:
# name: Alice
# age: 25
```

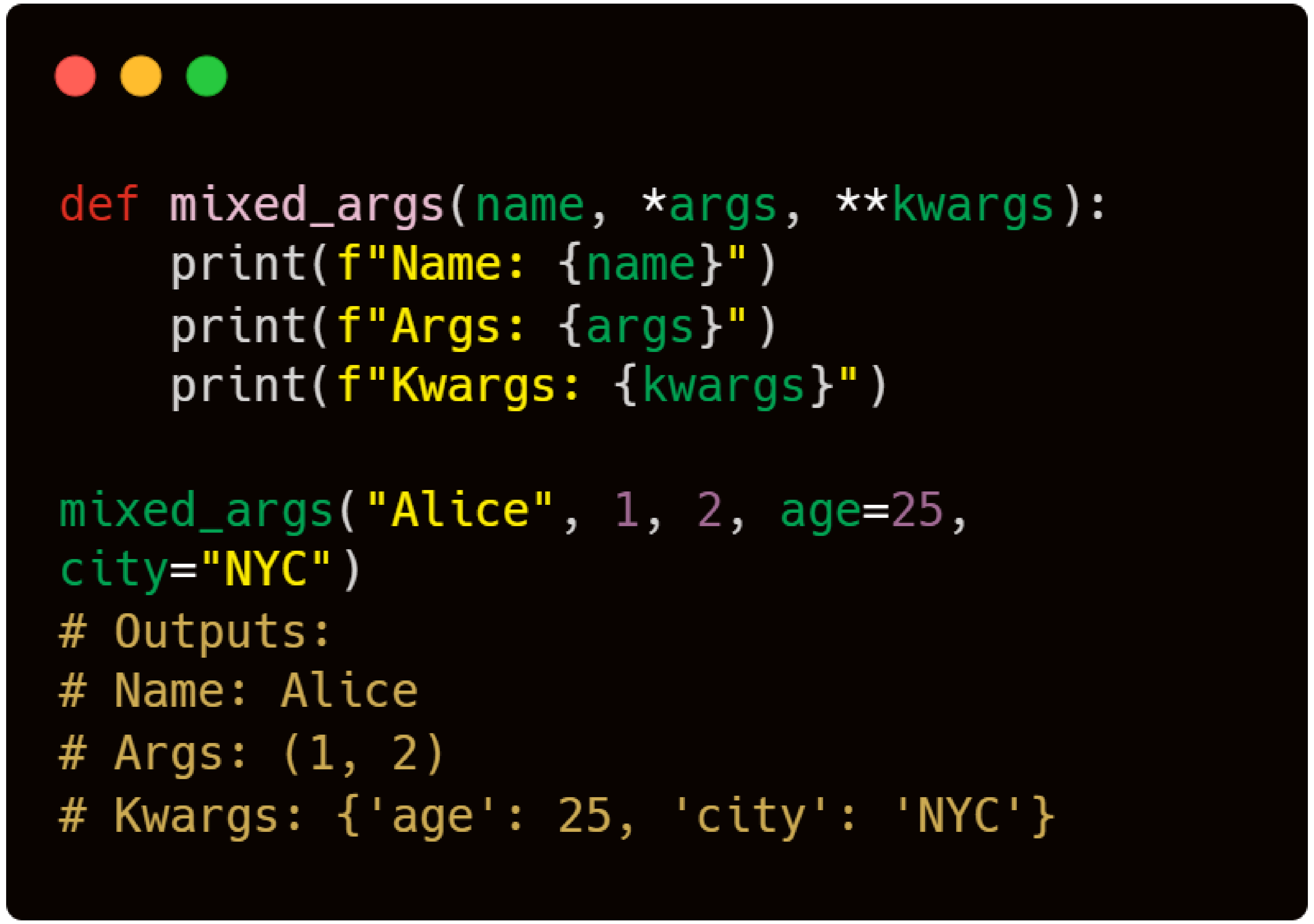
***args (Positional Arguments):**

- Collects additional positional arguments into a tuple.

****kwargs (Keyword Arguments):**

- Collects additional keyword arguments into a dictionary.

Positional & Keyword Arguments



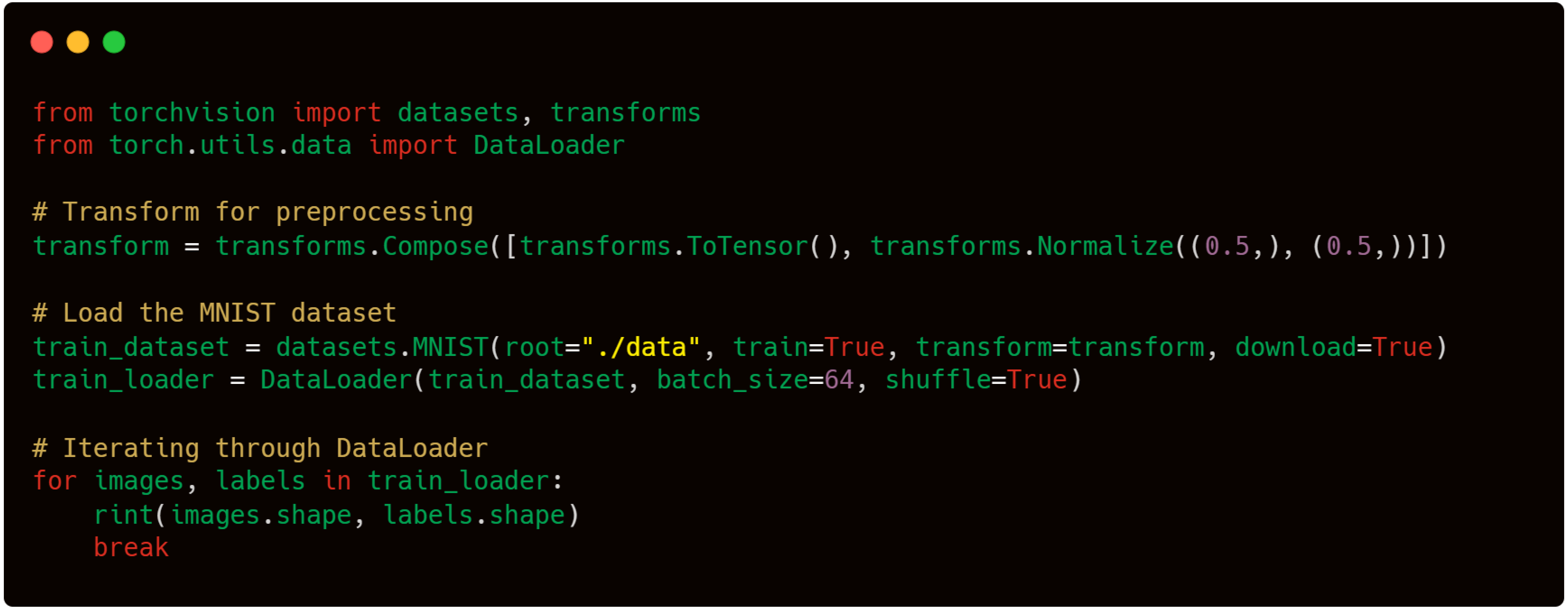
```
def mixed_args(name, *args, **kwargs):
    print(f"Name: {name}")
    print(f"Args: {args}")
    print(f"Kwargs: {kwargs}")

mixed_args("Alice", 1, 2, age=25,
city="NYC")
# Outputs:
# Name: Alice
# Args: (1, 2)
# Kwargs: {'age': 25, 'city': 'NYC'}
```

Key Differences:

- *args handles extra positional arguments, passed as a tuple.
- **kwargs handles extra keyword arguments, passed as a dictionary.

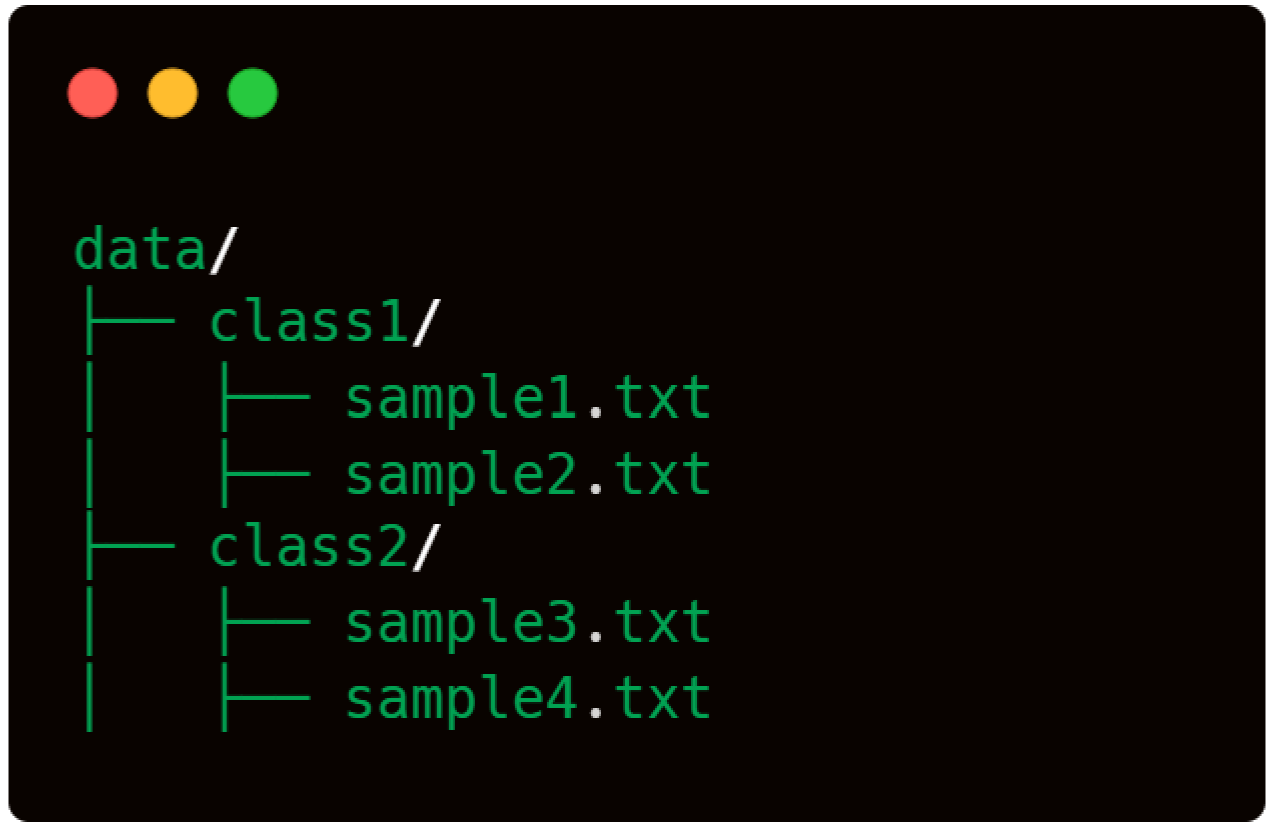
PyTorch Framework

Dataset & DataLoader

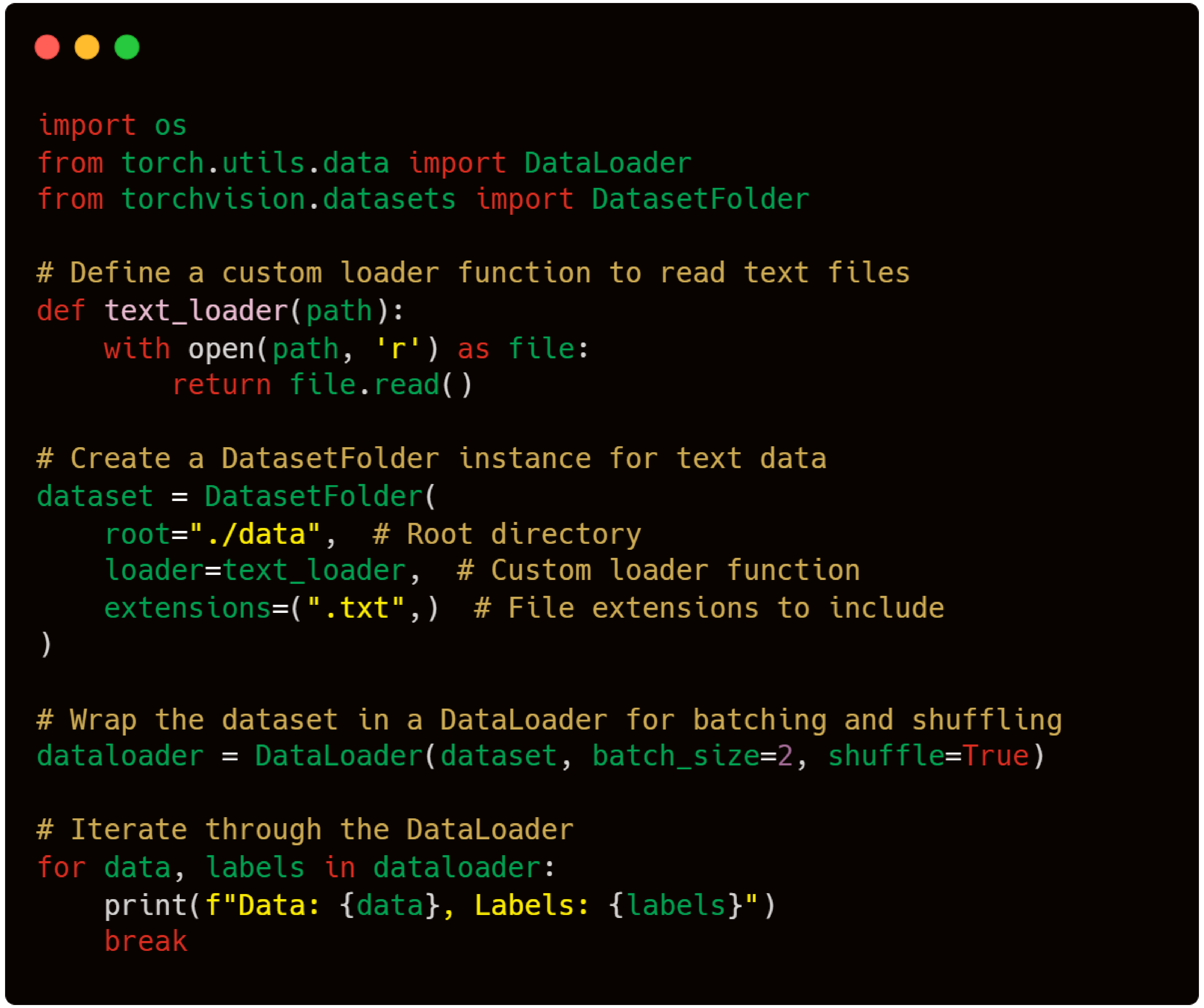


```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Transform for preprocessing
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

# Load the MNIST dataset
train_dataset = datasets.MNIST(root="./data", train=True, transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

# Iterating through DataLoader
for images, labels in train_loader:
    rint(images.shape, labels.shape)
    break
```

Built-in Dataset Classes

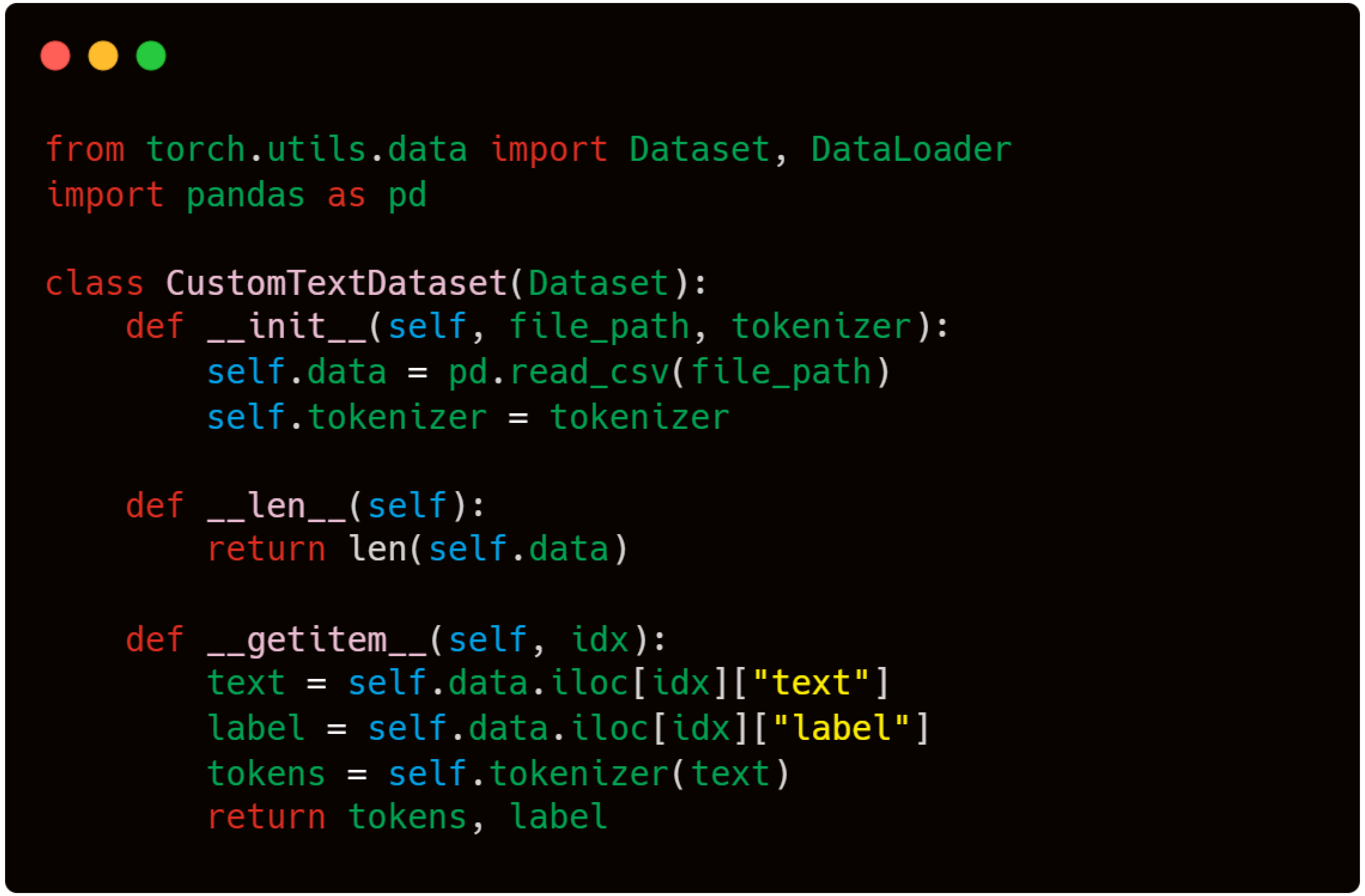


```
data/
├── class1/
│   ├── sample1.txt
│   ├── sample2.txt
├── class2/
│   ├── sample3.txt
│   ├── sample4.txt
```



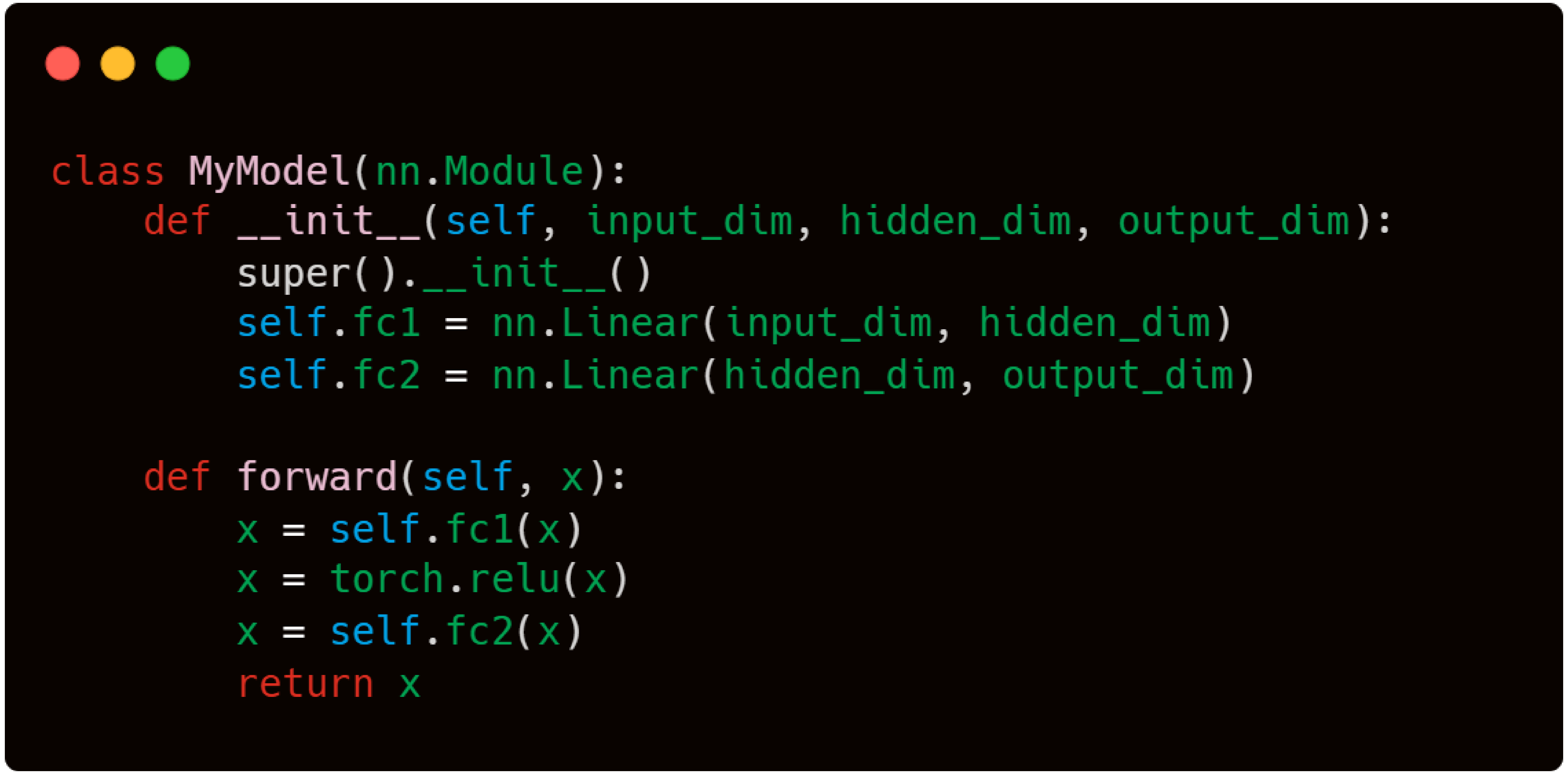
```python
import os
from torch.utils.data import DataLoader
from torchvision.datasets import DatasetFolder

# Define a custom loader function to read text files
def text_loader(path):
    with open(path, 'r') as file:
        return file.read()

# Create a DatasetFolder instance for text data
dataset = DatasetFolder(
    root="./data",  # Root directory
    loader=text_loader,  # Custom loader function
    extensions=(".txt",)  # File extensions to include
)

# Wrap the dataset in a DataLoader for batching and shuffling
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)

# Iterate through the DataLoader
for data, labels in dataloader:
    print(f"Data: {data}, Labels: {labels}")
    break
```

Custom Datasets



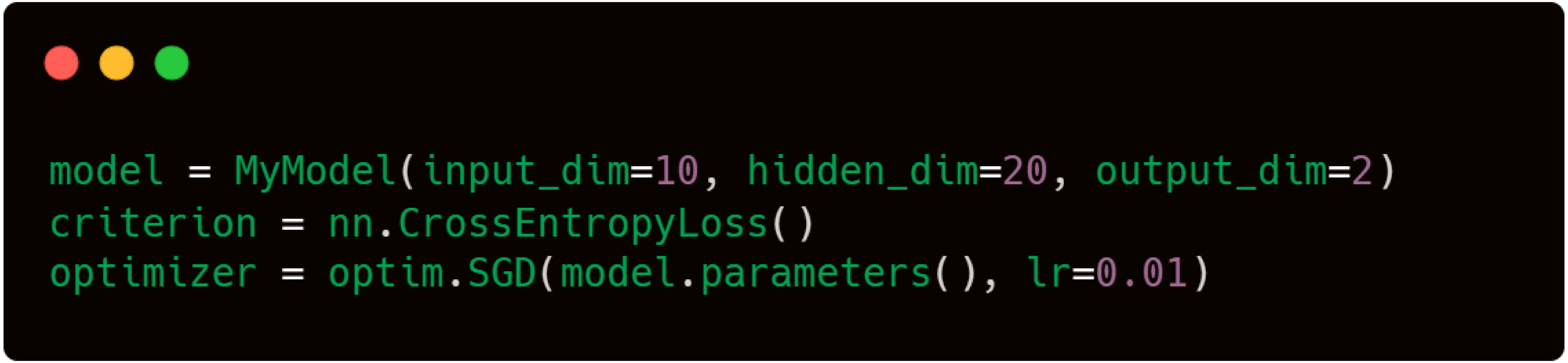
```python
from torch.utils.data import Dataset, DataLoader
import pandas as pd

class CustomTextDataset(Dataset):
    def __init__(self, file_path, tokenizer):
        self.data = pd.read_csv(file_path)
        self.tokenizer = tokenizer

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

    def __getitem__(self, idx):
        text = self.data.iloc[idx]["text"]
        label = self.data.iloc[idx]["label"]
        tokens = self.tokenizer(text)
        return tokens, label
```

Model Definition



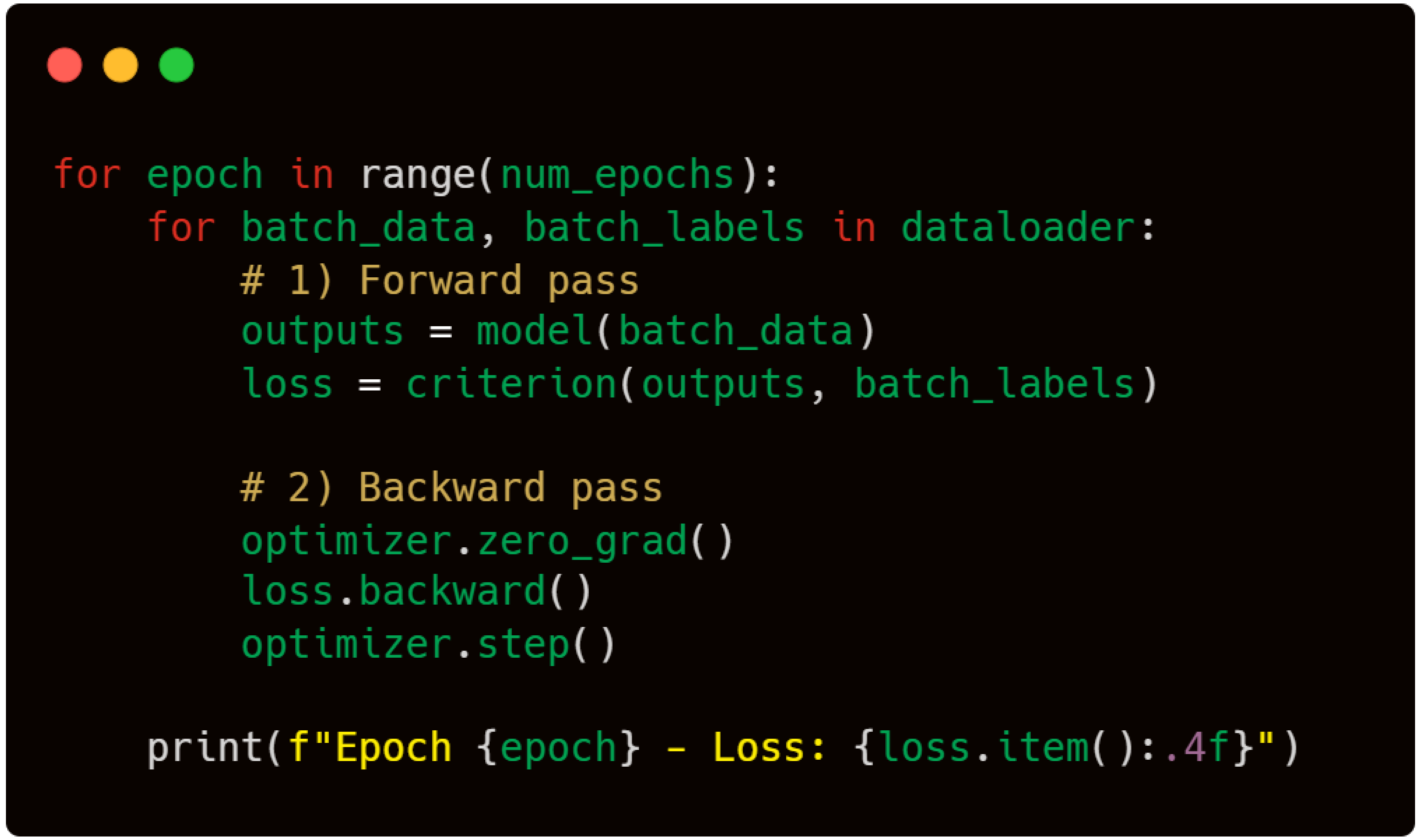
```python
class MyModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        return x
```

Loss & Optimizer



```python
model = MyModel(input_dim=10, hidden_dim=20, output_dim=2)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

Training Loop



```python
for epoch in range(num_epochs):
    for batch_data, batch_labels in dataloader:
        # 1) Forward pass
        outputs = model(batch_data)
        loss = criterion(outputs, batch_labels)

        # 2) Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch} - Loss: {loss.item():.4f}")
```

Step by step:

- **Get data:** from dataloader.
- **Forward pass:** model(batch_data).
- **Compute loss:** criterion(...).
- **Zero gradients:** optimizer.zero_grad()
- **Backward:** loss.backward() calculates gradients.
- **Update:** optimizer.step() updates parameters.

HuggingFace Ecosystem

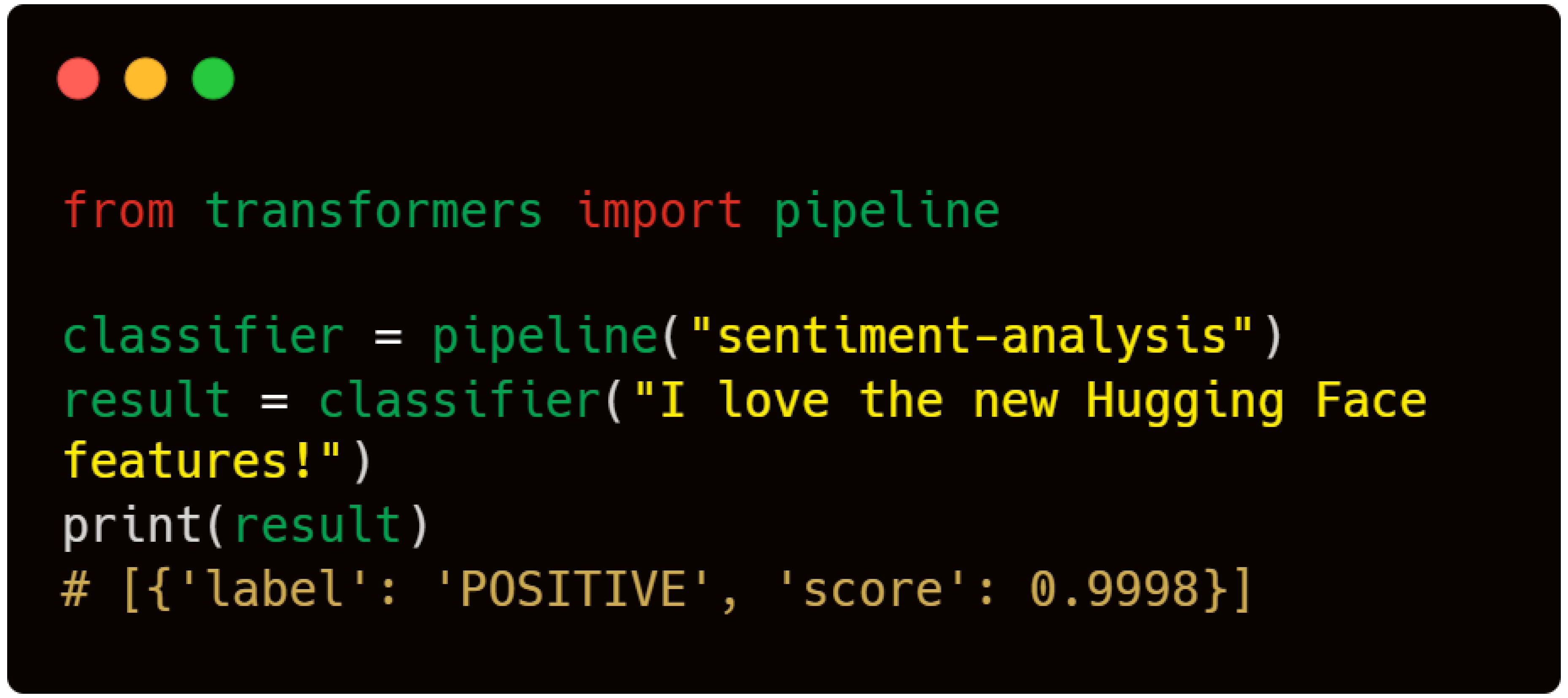
HuggingFace

- Platform for open-source Machine Learning
- Large hub of **pretrained models** and **datasets**
- **Community-driven** approach to sharing and collaboration
- Accessible libraries for **NLP, Computer Vision, and beyond**

Ecosystem

- **Transformers**
- **Datasets**
- **Tokenizers**
- **Diffusers**
- **Accelerate**
- **PEFT** (Parameter-Efficient Finetuning)
- **TRL** (Transformer Reinforcement Learning)

Transformers: Pipeline

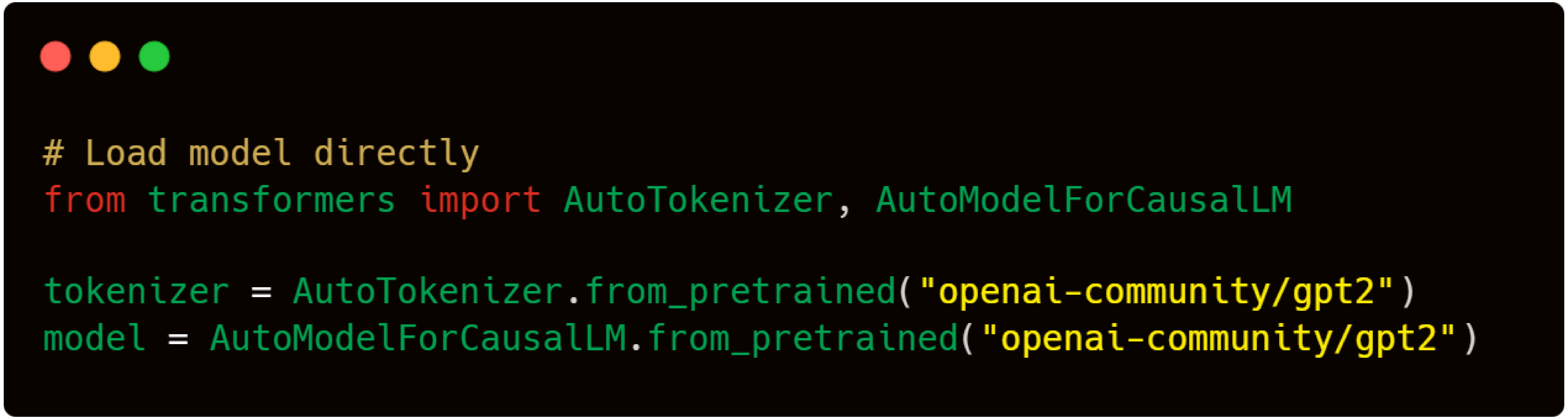


```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")
result = classifier("I love the new Hugging Face
features!")
print(result)
# [{'label': 'POSITIVE', 'score': 0.9998}]
```

Available Tasks:

- sentiment-analysis
- text-generation
- question-answering
- fill-mask
- Summarization
- translation

Load a Specific Model



```
# Load model directly
from transformers import AutoTokenizer, AutoModelForCausalLM

tokenizer = AutoTokenizer.from_pretrained("openai-community/gpt2")
model = AutoModelForCausalLM.from_pretrained("openai-community/gpt2")
```

AutoClasses

- **AutoModel:** Loads the base transformer model without any specific head.
- **AutoModelForSequenceClassification:** Loads a model with a classification head, suitable for tasks like sentiment analysis.
- **AutoModelForTokenClassification:** Loads a model with a token classification head, used for tasks like named entity recognition (NER).
- **AutoModelForQuestionAnswering:** Loads a model with a question-answering head, designed for extracting answers from passages.
- **AutoModelForMaskedLM:** Loads a model with a masked language modeling head, used for tasks like filling in missing words.
- **AutoModelForCausalLM:** Loads a model with a causal language modeling head, suitable for text generation tasks.
- **AutoModelForSeq2SeqLM:** Loads a sequence-to-sequence model with a language modeling head, used for tasks like translation and summarization.

Thank you!

Brighter World

McMaster University