NUMPY AND PYTORCH

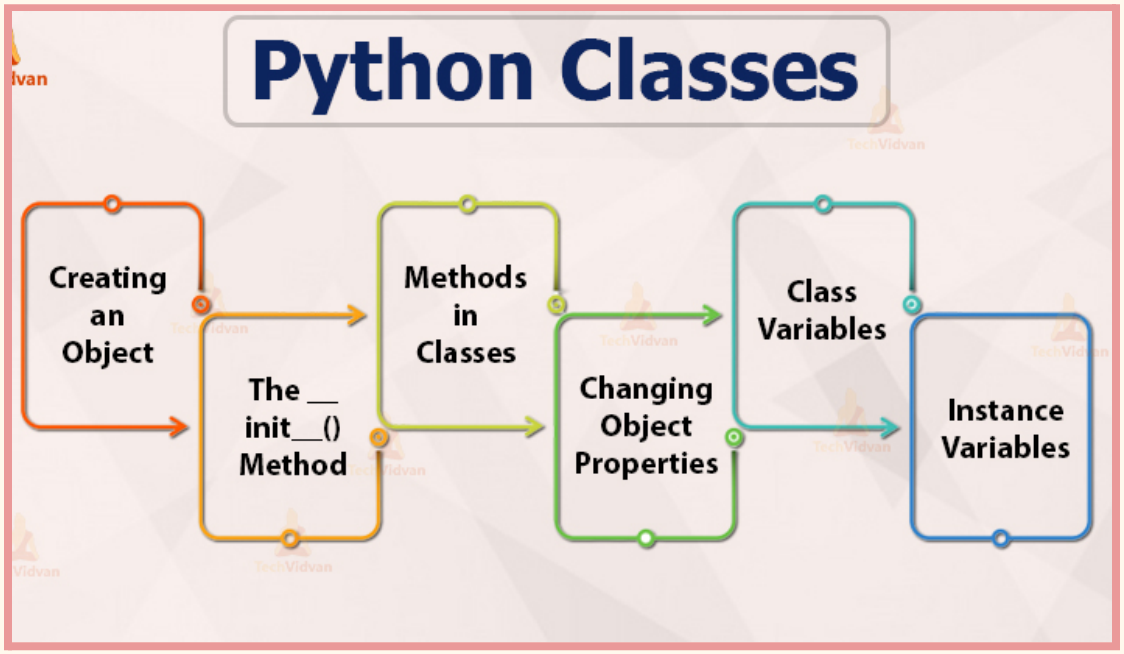
—

Objects, Classes and Methods

Python is an object-oriented programming language. Unlike procedure-oriented programming, where the main emphasis is on functions, object-oriented programming stresses on objects.

Class: A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type

Object: It is an instance of a class. An object in OOPS is nothing but a self-contained component which consists of methods and properties to make a particular type of data useful.



An object is also called an instance of a class and the process of creating this object is called instantiation.

Methods: Inside classes, you can define functions or methods that are part of this class **Constructors:** These are used for initializing new objects.

The "**init**" is called the initializer. It is known as a **constructor** in OOP concepts. This **init** method is automatically called when we instantiate the class. Its job is to make sure the class has any attributes it needs. It's sometimes also used to make sure that the object is in a **valid state** when it's instantiated
Classes contain characteristics called **Attributes**. We make a distinction between **instance attributes** and **class attributes**.

Instance Attributes are unique to each object, (an instance is another name for an object). Here, any Dog object we create will be able to store its name and age. We can change either attribute of either dog, without affecting any other dog objects we've created.

Inheritance:

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

It provides **reusability** of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.

When you add the **init**() function, the child class will no longer inherit the parent's **init**() function. The child's **init**() function overrides the inheritance of the parent's **init**() function.

To keep the inheritance of the parent's **init**() function, add a call to the parent's **init**() function:

```
class Student(Person):
```

```
def __init__(self, fname, lname):
Person.__init__(self, fname, lname)
```

Fundamentals of Numpy & PyTorch

NumPy arrays are used to store lists of numerical data and to represent vectors, matrices, and even tensors. NumPy arrays are designed to handle large data sets efficiently and with a minimum of fuss.

PyTorch is an open-source machine learning library used primarily for applications such as computer vision and natural language processing. PyTorch is a strong player in the field of deep learning and artificial intelligence, and it can be considered primarily as a research first library

Tensors are a type of data structure used in linear algebra, and like vectors and matrices, you can calculate arithmetic operations with tensors. Tensor is a multi-dimensional matrix containing elements of a single data type.

Interconversions

List --> Numpy Array

numpy.array(list)

List --> Torch Tensor

torch.tensor(list)

Numpy Array --> Torch Tensor

torch.from_numpy(numpy_array)

Torch Tensor --> Numpy Array

new_nparray = tensor1.detach().numpy()

Dot Product

NUMPY: np.dot(x,y)

PYTORCH: torch.dot(x,y)

Outer Product
NUMPY: np.outer(x,y)

PYTORCH: torch.outer(x, y)

Hadamard Product
NUMPY: x * y

PYTORCH: torch.mul(x, y)

Sum-Product
NUMPY: np.sum(np.outer(x,y))

PYTORCH: torch.sum(torch.outer(x, y))

ReLU

NUMPY: np.maximum(x, 0)

PYTORCH: torch.relu(x)

Prime ReLU

NUMPY: np.where(x > 0, 1, 0)
PYTORCH: torch.where(x > 0, 1, 0)

Data Loaders

Data Loader is a client application for the bulk import or export of **data**. Use it to insert, update, delete, or export records. When importing **data**, **Data Loader** reads, extracts, and loads **data** from comma-separated values (CSV) files or from a database connection.

Implementing a Dataset Class

In PyTorch, data loading utility is the torch.utils.data.DataLoader class. It represents a Python iterable over a dataset.

Ex:

```
class ExampleDataset1(torch.utils.data.Dataset):
```

We are required to implement three methods and can optionally add other methods depending on source data. The required methods are __init__(), __len__(), and __getitem__().

__init__()

__init__() is a builtin **function in Python**, that is called whenever an object is created. **__init__**() initializes the state for the object. Meaning, it is a place where we can set out initial or primary state of our object.

The **self-keyword** in **Python** is used to all the instances in a class. By using the **self keyword**, one can easily access all the instances defined within a class, including its methods and attributes.

Example:

```
def __init__ (self, X):
 ## Assign data to self
 self.X = X
 ## Assign length to self
 self.length = len(X)
```

__len__()

The len() function returns the number of items in an object.
When the object is a string, the len() function returns the number of characters in the string.

```
def __len__(self):
## Return length
return len(self.X)
```

__getitem__()

A method in Python, which when used in a class, allows its instances to use the [] (indexer) operators. Say x is an instance of this class, then x[i] is roughly equivalent to type(x).__getitem__(x, i).

The method __getitem__(self, key) defines behavior for when an item is accessed, using the notation self[key]. This is also part of both the mutable and immutable container protocols.

Example :

```
def __getitem__(self, i):
 ## Return data at index i
 return self.X[i]
```

- batch_size, which denotes the number of samples contained in each generated batch.
- shuffle. If set to True, we will get a new order of exploration at each pass (or just keep a linear exploration scheme otherwise). Shuffling the order in which examples are fed to the classifier is helpful so that batches between epochs do not look alike. Doing so will eventually make our model more robust.

Example:

```
dataloader1 = torch.utils.data.DataLoader(dataset1, batch_size=2,
shuffle=False, collate_fn=ExampleDataset1.collate_fn)
```

Working with collate_fn

collate_fn is called with a list of data samples at each time. It is expected to collate the input samples into a batch for yielding from the data loader iterator.

For instance, if each data sample consists of a 3-channel image and an integral class label, i.e., each element of the dataset returns a tuple (image, class_index), the default collate_fn collates a list of such tuples into a single tuple of a batched

image tensor and a batched class label Tensor. In particular, the default collate_fn has the following properties:

- It always prepends a new dimension as the batch dimension.
- It automatically converts NumPy arrays and Python numerical values into PyTorch Tensors.
- It preserves the data structure, e.g., if each sample is a dictionary, it outputs a dictionary with the same set of keys but batched Tensors as values (or lists if the values cannot be converted into Tensors). Same for list s, tuple s, namedtuple s, etc.

Example :

```
def collate_fn(batch):


## Convert batch to tensor (1 line)
batch_x = torch.as_tensor(batch)


## Return batched data and labels (1 line)
return (batch_x)
```