The `datapipes` folder holds the implementation of the `IterDataPipe` and `MapDataPipe`.

This document serves as an entry point for DataPipe implementation.

Implementing DataPipe

For the sake of an example, let us implement an `IterDataPipe` to apply a callable over data under `iter`. For `MapDataPipe`, please take reference from files in map folder and implement the corresponding `__getitem__` method.

Naming

The naming convention for DataPipe is Operation-er and with suffix of `IterDataPipe` because each DataPipe behaves like a container to apply the operation to data yielded from the source DataPipe. And, when importing the DataPipe into `iter` module under `datapipes`, each DataPipe will be aliased as Op-er without the suffix of `IterDataPipe`. Please check `__init__.py` in `iter` module for how we aliasing each DataPipe class. Like the example of `IterDataPipe` to map a function, we are going to name it as `MapperIterDataPipe` and alias it as `iter.Mapper` under `datapipes`.

Constructor

As DataSet now constructed by a stack of DataPipe-s, each DataPipe normally takes a source DataPipe as the first argument.

```
class MapperIterDataPipe(IterDataPipe):
    def __init__(self, dp, fn):
        super().__init__()
        self.dp = dp
        self.fn = fn
```

Note: - Avoid loading data from the source DataPipe in `__init__` function, in order to support lazy data loading and save memory. - If `IterDataPipe` instance holds data in memory, please be ware of the in-place modification of data. When second iterator is created from the instance, the data may have already changed. Please take `IterableWrapper` class as reference to `deepcopy` data for each iterator.

Iterator

For `IterDataPipe`, an `__iter__` function is needed to consume data from the source `IterDataPipe` then apply operation over the data before yield.

```
class MapperIterDataPipe(IterDataPipe):
    ...

    def __iter__(self):
```

```
        for d in self.dp:
            yield self.fn(d)
```

Length

In the most common cases, as the example of `MapperIterDataPipe` above, the `__len__` method of DataPipe should return the length of source DataPipe.

```
class MapperIterDataPipe(IterDataPipe):
    ...

    def __len__(self):
        return len(self.dp)
```

Note that `__len__` method is optional for `IterDataPipe`. Like `CSVParserIterDataPipe` in the Using DataPipe sector, `__len__` is not implemented because the size of each file streams is unknown for us before loading it.

Besides, in some special cases, `__len__` method can be provided, but it would either return an integer length or raise Error depending on the arguments of DataPipe. And, the Error is required to be `TypeError` to support Python's build-in functions like `list(dp)`. Please check NOTE [Lack of Default `__len__` in Python Abstract Base Classes] for detailed reason in PyTorch.

Registering DataPipe with functional API

Each DataPipe can be registered to support functional API using the decorator `functional_datapipe`.

```
@functional_datapipe("map")
class MapperIterDataPipe(IterDataPipe):
    ...
```

Then, the stack of DataPipe can be constructed in functional-programming manner.

```
>>> import torch.utils.data.datapipes as dp
>>> datapipes1 = dp.iter.FileOpener(['a.file', 'b.file']).map(fn=decoder).shuffle().batch(2)

>>> datapipes2 = dp.iter.FileOpener(['a.file', 'b.file'])
>>> datapipes2 = dp.iter.Mapper(datapipes2)
>>> datapipes2 = dp.iter.Shuffler(datapipes2)
>>> datapipes2 = dp.iter.Batcher(datapipes2, 2)
```

In the above example, `datapipes1` and `datapipes2` represent the exact same stack of `IterDataPipe`-s.

Using DataPipe

For example, we want to load data from CSV files with the following data pipeline: - List all csv files - Load csv files - Parse csv file and yield rows

To support the above pipeline, `CSVParser` is registered as `parse_csv_files` to consume file streams and expand them as rows.

```
@functional_datapipe("parse_csv_files")
class CSVParserIterDataPipe(IterDataPipe):
    def __init__(self, dp, **fmtparams):
        self.dp = dp
        self.fmtparams = fmtparams

    def __iter__(self):
        for filename, stream in self.dp:
            reader = csv.reader(stream, **self.fmtparams)
            for row in reader:
                yield filename, row
```

Then, the pipeline can be assembled as following:

```
>>> import torch.utils.data.datapipes as dp

>>> FOLDER = 'path/2/csv/folder'
>>> datapipe = dp.iter.FileLister([FOLDER]).filter(fn=lambda filename: filename.endswith('.c
>>> datapipe = dp.iter.FileOpener(datapipe, mode='rt')
>>> datapipe = datapipe.parse_csv_files(delimiter=' ')

>>> for d in datapipe: # Start loading data
...     pass
```