



Load

Your data can be stored in various places; they can be on your local machine's disk, in a Github repository, and in in-memory data structures like Python dictionaries and Pandas DataFrames. Wherever a dataset is stored, 🤖 Datasets can help you load it.

This guide will show you how to load a dataset from:

- The Hugging Face Hub
- Local files
- In-memory data
- Offline
- A specific slice of a split

For more details specific to loading other dataset modalities, take a look at the [load audio dataset guide](#), the [load image dataset guide](#), the [load video dataset guide](#), or the [load text dataset guide](#).

Hugging Face Hub

You can also load a dataset from any dataset repository on the Hub! Begin by [creating a dataset repository](#) and upload your data files. Now you can use the `load_dataset()` function to load the dataset.

For example, try loading the files from this [demo repository](#) by providing the repository namespace and dataset name. This dataset repository contains CSV files, and the code below loads the dataset from the CSV files:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset("lhoestq/demo1")
```

Some datasets may have more than one version based on Git tags, branches, or commits. Use the `revision` parameter to specify the dataset version you want to load:

```
>>> dataset = load_dataset(  
...     "lhoestq/custom_squad",  
...     revision="main" # tag name, or branch name, or commit hash  
... )
```

[!TIP]

Refer to the [Upload a dataset to the Hub](#) tutorial for more details on how to create a dataset repository on the Hub, and how to upload your data files.

A dataset loads by default all the data into the `train` split, or checks for mentions or split names in the data files names (e.g. "train", "test" and "validation"). Use the `data_files` parameter to map data files to splits like `train`, `validation` and `test`:

```
>>> data_files = {"train": "train.csv", "test": "test.csv"}  
>>> dataset = load_dataset("namespace/your_dataset_name", data_files=data_files)
```

[!WARNING]

If you don't specify which data files to use, `load_dataset()` will return all the data files. This can take a long time if you load a large dataset like C4, which is approximately 13TB of data.

You can also load a specific subset of the files with the `data_files` or `data_dir` parameter. These parameters can accept a relative path which resolves to the base path corresponding to where the dataset is loaded from.

```
>>> from datasets import load_dataset  
  
# load files that match the grep pattern  
>>> c4_subset = load_dataset("allenai/c4", data_files="en/c4-train.0000*-of-01024.json.gz")  
  
# load dataset from the en directory on the Hub  
>>> c4_subset = load_dataset("allenai/c4", data_dir="en")
```


The `split` parameter can also map a data file to a specific split:

```
>>> data_files = {"validation": "en/c4-validation.*.json.gz"}
>>> c4_validation = load_dataset("allenai/c4", data_files=data_files, split="validation")
```

Local and remote files

Datasets can be loaded from local files stored on your computer and from remote files. The datasets are most likely stored as a `csv`, `json`, `txt` or `parquet` file. The `load_dataset()` function can load each of these file types.

CSV

 Datasets can read a dataset made up of one or several CSV files (in this case, pass your CSV files as a list):

```
>>> from datasets import load_dataset
>>> dataset = load_dataset("csv", data_files="my_file.csv")
```

[!TIP]

For more details, check out the [how to load tabular datasets from CSV files](#) guide.

JSON

JSON files are loaded directly with `load_dataset()` as shown below:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset("json", data_files="my_file.json")
```

JSON files have diverse formats, but we think the most efficient format is to have multiple JSON objects; each line represents an individual row of data. For example:

```
{"a": 1, "b": 2.0, "c": "foo", "d": false}
{"a": 4, "b": -5.5, "c": null, "d": true}
```

Another JSON format you may encounter is a nested field, in which case you'll need to specify the `field` argument as shown in the following:

```

{"version": "0.1.0",
 "data": [{ "a": 1, "b": 2.0, "c": "foo", "d": false},
           {"a": 4, "b": -5.5, "c": null, "d": true}]
}

>>> from datasets import load_dataset
>>> dataset = load_dataset("json", data_files="my_file.json", field="data")

```

To load remote JSON files via HTTP, pass the URLs instead:

```

>>> base_url = "https://rajpurkar.github.io/SQuAD-explorer/dataset/"
>>> dataset = load_dataset("json", data_files={"train": base_url + "train-v1.1.json", "validation":

```

While these are the most common JSON formats, you'll see other datasets that are formatted differently. 🤖 Datasets recognizes these other formats and will fallback accordingly on the Python JSON loading methods to handle them.

Parquet

Parquet files are stored in a columnar format, unlike row-based files like a CSV. Large datasets may be stored in a Parquet file because it is more efficient and faster at returning your query.

To load a Parquet file:

```

>>> from datasets import load_dataset
>>> dataset = load_dataset("parquet", data_files={'train': 'train.parquet', 'test': 'test.parquet'})

```

To load remote Parquet files via HTTP, pass the URLs instead:

```

>>> base_url = "https://huggingface.co/datasets/wikimedia/wikipedia/resolve/main/20231101.ab/"
>>> data_files = {"train": base_url + "train-00000-of-00001.parquet"}
>>> wiki = load_dataset("parquet", data_files=data_files, split="train")

```

Arrow

Arrow files are stored in an in-memory columnar format, unlike row-based formats like CSV

and uncompressed formats like Parquet.

To load an Arrow file:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset("arrow", data_files={'train': 'train.arrow', 'test': 'test.arrow'})
```

To load remote Arrow files via HTTP, pass the URLs instead:

```
>>> base_url = "https://huggingface.co/datasets/croissantllm/croissant_dataset/resolve/main/english
>>> data_files = {"train": base_url + "train/data-00000-of-00080.arrow"}
>>> wiki = load_dataset("arrow", data_files=data_files, split="train")
```

Arrow is the file format used by 🤗 Datasets under the hood, therefore you can load a local Arrow file using `Dataset.from_file()` directly:

```
>>> from datasets import Dataset
>>> dataset = Dataset.from_file("data.arrow")
```

Unlike `load_dataset()`, `Dataset.from_file()` memory maps the Arrow file without preparing the dataset in the cache, saving you disk space.

The cache directory to store intermediate processing results will be the Arrow file directory in that case.

For now only the Arrow streaming format is supported. The Arrow IPC file format (also known as Feather V2) is not supported.

HDF5 files

HDF5 files are commonly used for storing large amounts of numerical data in scientific computing and machine learning. Loading HDF5 files with 🤗 Datasets is similar to loading CSV files:

```
>>> from datasets import load_dataset
>>> dataset = load_dataset("hdf5", data_files="data.h5")
```

Note that the HDF5 loader assumes that the file has "tabular" structure, i.e. that all datasets in the file have (the same number of) rows on their first dimension.

SQL

Read database contents with `from_sql()` by specifying the URI to connect to your database. You can read both table names and queries:

```
>>> from datasets import Dataset
# load entire table
>>> dataset = Dataset.from_sql("data_table_name", con="sqlite:///sqlite_file.db")
# load from query
>>> dataset = Dataset.from_sql("SELECT text FROM table WHERE length(text) > 100 LIMIT 10", con="sq
```

[!TIP]

For more details, check out the [how to load tabular datasets from SQL databases](#) guide.

WebDataset

The [WebDataset](#) format is based on TAR archives and is suitable for big image datasets. Because of their size, WebDatasets are generally loaded in streaming mode (using `streaming=True`).

You can load a WebDataset like this:

```
>>> from datasets import load_dataset
>>>
>>> path = "path/to/train/*.tar"
>>> dataset = load_dataset("webdataset", data_files={"train": path}, split="train", streaming=True
```

To load remote WebDatasets via HTTP, pass the URLs instead:

```
>>> from datasets import load_dataset
>>>
>>> base_url = "https://huggingface.co/datasets/lhoestq/small-publaynet-wds/resolve/main/publaynet"
>>> urls = [base_url.format(i=i) for i in range(4)]
>>> dataset = load_dataset("webdataset", data_files={"train": urls}, split="train", streaming=True
```

Multiprocessing

When a dataset is made of several files (that we call "shards"), it is possible to significantly speed up the dataset downloading and preparation step.


You can choose how many processes you'd like to use to prepare a dataset in parallel using `num_proc`.

In this case, each process is given a subset of shards to prepare:

```
from datasets import load_dataset

imagenet = load_dataset("timm/imagenet-1k-wds", num_proc=8)
ml_librispeech_spanish = load_dataset("facebook/multilingual_librispeech", "spanish", num_proc=8)
```

In-memory data

 Datasets will also allow you to create a [Dataset](#) directly from in-memory data structures like Python dictionaries and Pandas DataFrames.

Python dictionary

Load Python dictionaries with `from_dict()`:

```
>>> from datasets import Dataset
>>> my_dict = {"a": [1, 2, 3]}
>>> dataset = Dataset.from_dict(my_dict)
```

Python list of dictionaries

Load a list of Python dictionaries with `from_list()`:

```
>>> from datasets import Dataset
>>> my_list = [{"a": 1}, {"a": 2}, {"a": 3}]
>>> dataset = Dataset.from_list(my_list)
```

Python generator

Create a dataset from a Python generator with `from_generator()`:

```
>>> from datasets import Dataset
>>> def my_gen():
...     for i in range(1, 4):
...         yield {"a": i}
...
>>> dataset = Dataset.from_generator(my_gen)
```

This approach supports loading data larger than available memory.

You can also define a sharded dataset by passing lists to `gen_kwargs`:

```
>>> def gen(shards):
...     for shard in shards:
...         with open(shard) as f:
...             for line in f:
...                 yield {"line": line}
...
>>> shards = [f"data{i}.txt" for i in range(32)]
>>> ds = IterableDataset.from_generator(gen, gen_kwargs={"shards": shards})
>>> ds = ds.shuffle(seed=42, buffer_size=10_000) # shuffles the shards order + uses a shuffle buffer
>>> from torch.utils.data import DataLoader
>>> dataloader = DataLoader(ds.with_format("torch"), num_workers=4) # give each worker a subset of shards
```

Pandas DataFrame

Load Pandas DataFrames with `from_pandas()`:

```
>>> from datasets import Dataset
>>> import pandas as pd
>>> df = pd.DataFrame({"a": [1, 2, 3]})
>>> dataset = Dataset.from_pandas(df)
```

[!TIP]

For more details, check out the [how to load tabular datasets from Pandas DataFrames](#)

guide.

Offline

Even if you don't have an internet connection, it is still possible to load a dataset. As long as you've downloaded a dataset from the Hub repository before, it should be cached. This means you can reload the dataset from the cache and use it offline.

If you know you won't have internet access, you can run 🤖 Datasets in full offline mode. This saves time because instead of waiting for the Dataset builder download to time out, 🤖 Datasets will look directly in the cache. Set the environment variable `HF_HUB_OFFLINE` to `1` to enable full offline mode.

Slice splits

You can also choose only to load specific slices of a split. There are two options for slicing a split: using strings or the [ReadInstruction](#) API. Strings are more compact and readable for simple cases, while [ReadInstruction](#) is easier to use with variable slicing parameters.

Concatenate a `train` and `test` split by:

```
>>> train_test_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split="train+test")
===STRINGAPI-READINSTRUCTION-SPLIT===
>>> ri = datasets.ReadInstruction("train") + datasets.ReadInstruction("test")
>>> train_test_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split=ri)
```

Select specific rows of the `train` split:

```
>>> train_10_20_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split="train")
===STRINGAPI-READINSTRUCTION-SPLIT===
>>> train_10_20_ds = datasets.load_dataset("bookcorpu", split=datasets.ReadInstruction("train", from_index=10, to_index=20))
```

Or select a percentage of a split with:

```
>>> train_10pct_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split="train",
===STRINGAPI-READINSTRUCTION-SPLIT===
>>> train_10_20_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split=train_10_20_ds)
```

Select a combination of percentages from each split:

```
>>> train_10_80pct_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split="train",
===STRINGAPI-READINSTRUCTION-SPLIT===
>>> ri = (datasets.ReadInstruction("train", to=10, unit="%") + datasets.ReadInstruction("train", from=10, to=80, unit="%"))
>>> train_10_80pct_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split=ri)
```

Finally, you can even create cross-validated splits. The example below creates 10-fold cross-validated splits. Each validation dataset is a 10% chunk, and the training dataset makes up the remaining complementary 90% chunk:

```
>>> val_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split=[f"train[{k}%:10%]" for k in range(10)])
>>> train_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split=[f"train[:{k}%]" for k in range(10)])
===STRINGAPI-READINSTRUCTION-SPLIT===
>>> val_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", [datasets.ReadInstruction("train", from=k*10, to=(k+1)*10, unit="%") for k in range(10)])
>>> train_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", [(datasets.ReadInstruction("train", to=k*10, unit="%") + datasets.ReadInstruction("train", from=k*10, to=90, unit="%")) for k in range(10)])
```

Percent slicing and rounding

The default behavior is to round the boundaries to the nearest integer for datasets where the requested slice boundaries do not divide evenly by 100. As shown below, some slices may contain more examples than others. For instance, if the following train split includes 999 records, then:

```
# 19 records, from 500 (included) to 519 (excluded).
>>> train_50_52_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split="train",
# 20 records, from 519 (included) to 539 (excluded).
>>> train_52_54_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split="train",
```

If you want equal sized splits, use `pct1_dropremainder` rounding instead. This treats the specified percentage boundaries as multiples of 1%.

```
# 18 records, from 450 (included) to 468 (excluded).
>>> train_50_52pct1_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split="train_50_52pct1")
# 18 records, from 468 (included) to 486 (excluded).
>>> train_52_54pct1_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split="train_52_54pct1")
# Or equivalently:
>>> train_50_52pct1_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split="train_50_52pct1")
>>> train_52_54pct1_ds = datasets.load_dataset("ajibawa-2023/General-Stories-Collection", split="train_52_54pct1")
```

[!WARNING]

`pct1_dropremainder` rounding may truncate the last examples in a dataset if the number of examples in your dataset don't divide evenly by 100.

Troubleshooting

Sometimes, you may get unexpected results when you load a dataset. Two of the most common issues you may encounter are manually downloading a dataset and specifying features of a dataset.

Specify features

When you create a dataset from local files, the [Features](#) are automatically inferred by [Apache Arrow](#). However, the dataset's features may not always align with your expectations, or you may want to define the features yourself. The following example shows how you can add custom labels with the [ClassLabel](#) feature.

Start by defining your own labels with the [Features](#) class:

```
>>> class_names = ["sadness", "joy", "love", "anger", "fear", "surprise"]
>>> emotion_features = Features({'text': Value('string'), 'label': ClassLabel(names=class_names)})
```

Next, specify the `features` parameter in [load_dataset\(\)](#) with the features you just created:

```
>>> dataset = load_dataset('csv', data_files=file_dict, delimiter=';', column_names=['text', 'label'])
```

Now when you look at your dataset features, you can see it uses the custom labels you defined:

```
>>> dataset['train'].features
{'text': Value('string'),
 'label': ClassLabel(names=['sadness', 'joy', 'love', 'anger', 'fear', 'surprise'])}
```