TensorBay

Graviti

Mar 25, 2021

QUICK START

As an expert in unstructured data management, TensorBay provides services like data hosting, complex data version management, online data visualization, and data collaboration. TensorBay's unified authority management makes your data sharing and collaborative use more secure.

This documentation describes *SDK* and *CLI* tools for using TensorBay.

CHAPTER

ONE

WHAT CAN TENSORBAY SDK DO?

TensorBay Python SDK is a python library to access TensorBay and manage your datasets. It provides:

- A *pythonic way* to access your TensorBay resources by TensorBay OpenAPI.
- An easy-to-use CLI tool *gas* (Graviti AI service) to communicate with TensorBay.
- A consistent *dataset structure* to read and write your datasets.

1.1 Getting started with TensorBay

1.1.1 Installation

To install TensorBay SDK and CLI by **pip**, run the following command:

```
$ pip3 install tensorbay
```

To verify the SDK and CLI version, run the following command:

```
$ gas --version
```

1.1.2 Registration

Before using TensorBay SDK, please finish the following registration steps:

- Please visit Graviti AI Service(GAS) to sign up.
- Please visit this page to get an AccessKey.

Note: An AccessKey is needed to authenticate identity when using TensorBay via SDK or CLI.

1.1.3 Usage

Authorize a Client Object

```
from tensorbay import GAS

gas = GAS("<YOUR_ACCESSKEY>")
```

See *this page* for details about authenticating identity via CLI.

Create a Dataset

```
gas.create_dataset("DatasetName")
```

List Dataset Names

```
dataset_list = list(gas.list_dataset_names())
```

Upload Images to the Dataset

```
from tensorbay.dataset import Data, Dataset

# Organize the local dataset by the "Dataset" class before uploading.
dataset = Dataset("DatasetName")

# TensorBay uses "segment" to separate different parts in a dataset.
segment = dataset.create_segment()

segment.append(Data("0000001.jpg"))
segment.append(Data("0000002.jpg"))

dataset_client = gas.upload_dataset(dataset)

# TensorBay provides dataset version control feature, commit the uploaded data before␣
↪using it.
dataset_client.commit("Initial commit")
```

Read Images from the Dataset

```
from PIL import Image
from tensorbay.dataset import Segment

dataset_client = gas.get_dataset("DatasetName")

segment = Segment("", dataset_client)

for data in segment:
    with data.open() as fp:
        image = Image.open(fp)
        width, height = image.size
        image.show()
```

Delete the Dataset

```
gas.delete_dataset("DatasetName")
```

1.2 Examples

We write examples for labels in *Label Format*.
Table. 1.1 lists the examples, including their data types and label types.

Table 1.1: Examples

Examples	Description
Dataset Management: Dogs vs Cats	This example describes how to manage Dogs vs Cats dataset, which is an image dataset with *Classification* label.
Dataset Management: 20 Newsgroups	This example describes how to manage 20 Newsgroups dataset, which is a text dataset with *Classification* label.
Dataset Management: BSTLD	This example describes how to manage BSTLD dataset, which is an image dataset with *Box2D* label.
Dataset Management: Neolix OD	This example describes how to manage Neolix OD dataset, which is a Point Cloud dataset with *Box3D* label.
Dataset Management: LeedsSportsPose	This example describes how to manage LeedsSportsPose dataset, which is an image dataset with *Keypoints2D* label.
Dataset Management: THCHS-30	This example describes how to manage THCHS-30 dataset, which is an audio dataset with *Sentence* label.
Read "Dataset" Class: BSTLD	This example describes how to read BSTLD dataset when it has been organized by a `Dataset` class.

1.2.1 Dogs vs Cats

This topic describes how to manage the "Dogs vs Cats" dataset.

"Dogs vs Cats" is a dataset with *Classification* label type. See this page for more details about this dataset.

Authorize a Client Object

First of all, create a GAS client.

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
```

Create Dataset

Then, create a dataset client by passing the dataset name to the GAS client.

```
gas.create_dataset("Dogs vs Cats")
```

List Dataset Names

To check if you have created "Dogs vs Cats" dataset, you can list all your available datasets. See *this page* for details.

```
list(gas.list_dataset_names())
```

Note: Note that method `list_dataset_names()` returns an iterator, use `list()` to transfer it to a "list".

Organize Dataset

Now we describe how to organize the "Dogs vs Cats" dataset by the *`Dataset`* object before uploading it to TensorBay. It takes the following steps to organize "Dogs vs Cats".

Write the Catalog

The first step is to write the *catalog*. Catalog is a json file contains all label information of one dataset. See *this page* for more details. The only annotation type for "Dogs vs Cats" is *Classification*, and there are 2 *Category* types.

```
{
    "CLASSIFICATION": {
        "categories": [{ "name": "cat" }, { "name": "dog" }]
    }
}
```

Write the Dataloader

The second step is to write the *dataloader*. The function of *dataloader* is to read the dataset into a `Dataset` object. The *code block* below displays the "Dogs vs Cats" dataloader.

```python
#!/usr/bin/env python3
#
# Copyright 2021 Graviti. Licensed under MIT License.
#
# pylint: disable=invalid-name

"""Dataloader of the DogsVsCats dataset."""

import os

from ...dataset import Data, Dataset
from ...label import Classification
from ..utility import glob

DATASET_NAME = "Dogs vs Cats"
_SEGMENTS = {"train": True, "test": False}


def DogsVsCats(path: str) -> Dataset:
    """Dataloader of the DogsVsCats dataset.

    Arguments:
        path: The root directory of the dataset.
            The file structure should be like::

                <path>
                    train/
                        cat.0.jpg
                        ...
                        dog.0.jpg
                        ...
                    test/
                        1000.jpg
                        1001.jpg
                        ...

    Returns:
        Loaded ``Dataset`` object.

    """
    root_path = os.path.abspath(os.path.expanduser(path))
    dataset = Dataset(DATASET_NAME)
    dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))

    for segment_name, is_labeled in _SEGMENTS.items():
        segment = dataset.create_segment(segment_name)
        image_paths = glob(os.path.join(root_path, segment_name, "*.jpg"))
        for image_path in image_paths:
            data = Data(image_path)
            if is_labeled:
                data.label.classification = Classification(os.path.basename(image_
↪path)[:3])
```

(continues on next page)

(continued from previous page)

```
52              segment.append(data)
53
54      return dataset
```

Note that after creating the *dataset*, you need to load the *catalog*.(L43) The catalog file "catalog.json" is in the same directory with dataloader file.

In this example, we create segments by `dataset.create_segment(SEGMENT_NAME)`. You can also create a default segment without giving a specific name, then its name will be "".

See *this page* for more details for about Classification annotation details.

Note: The *Dogs vs Cats dataloader* above uses relative import(L11-12). However, when you write your own dataloader you should use regular import. And when you want to contribute your own dataloader, remember to use relative import.

Upload Dataset

After you finish the *dataloader* and organize the "Dogs vs Cats" into a `Dataset` object, you can upload it to TensorBay for sharing, reuse, etc.

```
# dataset is the one you initialized in "Organize Dataset" section
dataset_client = gas.upload_dataset(dataset, jobs=8, skip_uploaded_files=False)
dataset_client.commit("Dogs vs Cats")
```

Remember to execute the commit step after uploading. If needed, you can re-upload and commit again. Please see *this page* for more details about version control.

Note: Commit operation can also be done on our GAS Platform.

Read Dataset

Now you can read "Dogs vs Cats" dataset from TensorBay.

```
dataset_client = gas.get_dataset("Dogs vs Cats")
```

In *dataset* "Dogs vs Cats", there are two *Segments*: `train` and `test`, you can get the segment names by list them all.

```
list(dataset_client.list_segment_names())
```

You can get a segment by passing the required segment name.

```
from tensorbay.dataset import Segment

train_segment = Segment("train", dataset_client)
```

In the train *segment*, there is a sequence of *data*. You can get one by index.

```
data = train_segment[0]
```

Note: If the *segment* or advanced_features/fusion_dataset/fusion_dataset_structure:fusion segment is created without given name, then its name will be "".

In each *data*, there is a sequence of *Classification* annotations. You can get one by index.

```
category = data.label.classification.category
```

There is only one label type in "Dogs vs Cats" dataset, which is `classification`. The information stored in *Category* is one of the category names in "categories" list of *catalog.json*. See *this page* for more details about the structure of Classification.

Delete Dataset

To delete "Dogs vs Cats", run the following code:

```
gas.delete_dataset("Dogs vs Cats")
```

1.2.2 BSTLD

This topic describes how to manage the "BSTLD" dataset.

"BSTLD" is a dataset with *Box2D* label type (Fig. 1.1). See this page for more details about this dataset.

Fig. 1.1: The preview of a cropped image with labels from "BSTLD".

Authorize a Client Object

First of all, create a GAS client.

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
```

Create Dataset

Then, create a dataset client by passing the dataset name to the GAS client.

```
gas.create_dataset("BSTLD")
```

List Dataset Names

To check if you have created "BSTLD" dataset, you can list all your available datasets. See *this page* for details.

```
list(gas.list_dataset_names())
```

Note: Note that method `list_dataset_names()` returns an iterator, use `list()` to transfer it to a "list".

Organize Dataset

Now we describe how to organize the "BSTLD" dataset by the *Dataset* object before uploading it to TensorBay. It takes the following steps to organize "BSTLD".

Write the Catalog

The first step is to write the *catalog*. Catalog is a json file contains all label information of one dataset. See *this page* for more details. The only annotation type for "BSTLD" is *Box2D*, and there are 13 *Category* types and one *Attributes* type.

```
{
    "BOX2D": {
        "categories": [
            { "name": "Red" },
            { "name": "RedLeft" },
            { "name": "RedRight" },
            { "name": "RedStraight" },
            { "name": "RedStraightLeft" },
            { "name": "Green" },
            { "name": "GreenLeft" },
            { "name": "GreenRight" },
            { "name": "GreenStraight" },
            { "name": "GreenStraightLeft" },
            { "name": "GreenStraigntRight" },
            { "name": "Yellow" },
            { "name": "off" }
```

(continues on next page)

(continued from previous page)

```
        ],
        "attributes": [
            {
                "name": "occluded",
                "type": "boolean"
            }
        ]
    }
}
```

Write the Dataloader

The second step is to write the *dataloader*. The function of *dataloader* is to read the dataset into a `Dataset` object. The *code block* below displays the "BSTLD" dataloader.

```
#!/usr/bin/env python3
#
# Copyright 2021 Graviti. Licensed under MIT License.
#
# pylint: disable=invalid-name

"""Dataloader of the BSTLD dataset."""

import os

from ...dataset import Data, Dataset
from ...label import LabeledBox2D

DATASET_NAME = "BSTLD"

_LABEL_FILENAME_DICT = {
    "test": "test.yaml",
    "train": "train.yaml",
    "additional": "additional_train.yaml",
}


def BSTLD(path: str) -> Dataset:
    """Dataloader of the BSTLD dataset.

    Arguments:
        path: The root directory of the dataset.
            The file structure should be like::

                <path>
                    rgb/
                        additional/
                            2015-10-05-10-52-01_bag/
                                <image_name>.jpg
                                ...
                            ...
                        test/
                            <image_name>.jpg
                            ...
                        train/
```

(continues on next page)

(continued from previous page)

```
                                2015-05-29-15-29-39_arastradero_traffic_light_loop_bag/
                                    <image_name>.jpg
                                    ...
                                ...
                        test.yaml
                        train.yaml
                        additional_train.yaml

    Returns:
        Loaded `Dataset` object.

    """
    import yaml  # pylint: disable=import-outside-toplevel

    root_path = os.path.abspath(os.path.expanduser(path))

    dataset = Dataset(DATASET_NAME)
    dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))

    for mode, label_file_name in _LABEL_FILENAME_DICT.items():
        segment = dataset.create_segment(mode)
        label_file_path = os.path.join(root_path, label_file_name)

        with open(label_file_path, encoding="utf-8") as fp:
            labels = yaml.load(fp, yaml.FullLoader)

        for label in labels:
            if mode == "test":
                # the path in test label file looks like:
                # /absolute/path/to/<image_name>.png
                file_path = os.path.join(root_path, "rgb", "test", label["path"].
↪rsplit("/", 1)[-1])
            else:
                # the path in label file looks like:
                # ./rgb/additional/2015-10-05-10-52-01_bag/<image_name>.png
                file_path = os.path.join(root_path, *label["path"][2:].split("/"))
            data = Data(file_path)
            data.label.box2d = [
                LabeledBox2D(
                    box["x_min"],
                    box["y_min"],
                    box["x_max"],
                    box["y_max"],
                    category=box["label"],
                    attributes={"occluded": box["occluded"]},
                )
                for box in label["boxes"]
            ]
            segment.append(data)

    return dataset
```

Note that after creating the *dataset*, you need to load the *catalog*.(L58) The catalog file "catalog.json" is in the same directory with dataloader file.

In this example, we create segments by `dataset.create_segment(SEGMENT_NAME)`. You can also create a default segment without giving a specific name, then its name will be "".

See *this page* for more details for about Box2D annotation details.

Note: The *BSTLD dataloader* above uses relative import(L11-12). However, when you write your own dataloader you should use regular import. And when you want to contribute your own dataloader, remember to use relative import.

Upload Dataset

After you finish the *dataloader* and organize the "BSTLD" into a `Dataset` object, you can upload it to TensorBay for sharing, reuse, etc.

```
# dataset is the one you initialized in "Organize Dataset" section
dataset_client = gas.upload_dataset(dataset, jobs=8, skip_uploaded_files=False)
dataset_client.commit("BSTLD")
```

Remember to execute the commit step after uploading. If needed, you can re-upload and commit again. Please see *this page* for more details about version control.

Note: Commit operation can also be done on our GAS Platform.

Read Dataset

Now you can read "BSTLD" dataset from TensorBay.

```
dataset_client = gas.get_dataset("BSTLD")
```

In *dataset* "BSTLD", there are three *Segments*: `train`, `test` and `additional`, you can get the segment names by list them all.

```
list(dataset_client.list_segment_names())
```

You can get a segment by passing the required segment name.

```
from tensorbay.dataset import Segment

train_segment = Segment("train", dataset_client)
```

In the train *segment*, there is a sequence of *data*. You can get one by index.

```
data = train_segment[3]
```

Note: If the *segment* or advanced_features/fusion_dataset/fusion_dataset_structure:fusion segment is created without given name, then its name will be "".

In each *data*, there is a sequence of *Box2D* annotations. You can get one by index.

```
label_box2d = data.label.box2d[0]
category = label_box2d.category
attributes = label_box2d.attributes
```

There is only one label type in "BSTLD" dataset, which is `box2d`. The information stored in *Category* is one of the category names in "categories" list of *catalog.json*. The information stored in *Attributes* is one of the attributes in "attributes" list of *catalog.json*. See *this page* for more details about the structure of Box2D.

Delete Dataset

To delete "BSTLD", run the following code:

```
gas.delete_dataset("BSTLD")
```

1.2.3 LeedsSportsPose

This topic describes how to manage the "LeedsSportsPose" dataset.

"LeedsSportsPose" is a dataset with *Keypoints2D* label type (Fig. 1.2). See this page for more details about this dataset.

Authorize a Client Object

First of all, create a GAS client.

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
```

Create Dataset

Then, create a dataset client by passing the dataset name to the GAS client.

```
gas.create_dataset("LeedsSportsPose")
```

List Dataset Names

To check if you have created "LeedsSportsPose" dataset, you can list all your available datasets. See *this page* for details.

```
list(gas.list_dataset_names())
```

Note: Note that method `list_dataset_names()` returns an iterator, use `list()` to transfer it to a "list".

Fig. 1.2: The preview of an image with labels from “LeedsSportsPose”.

Organize Dataset

Now we describe how to organize the "LeedsSportsPose" dataset by the `Dataset` object before uploading it to TensorBay. It takes the following steps to organize "LeedsSportsPose".

Write the Catalog

The first step is to write the *catalog*. Catalog is a json file contains all label information of one dataset. See *this page* for more details. The only annotation type for "LeedsSportsPose" is *Keypoints2D*.

```
{
    "KEYPOINTS2D": {
        "keypoints": [
            {
                "number": 14,
                "names": [
                    "Right ankle",
                    "Right knee",
                    "Right hip",
                    "Left hip",
                    "Left knee",
                    "Left ankle",
                    "Right wrist",
                    "Right elbow",
                    "Right shoulder",
                    "Left shoulder",
                    "Left elbow",
                    "Left wrist",
                    "Neck",
                    "Head top"
                ],
                "skeleton": [
                    [0, 1],
                    [1, 2],
                    [3, 4],
                    [4, 5],
                    [6, 7],
                    [7, 8],
                    [9, 10],
                    [10, 11],
                    [12, 13],
                    [12, 2],
                    [12, 3]
                ],
                "visible": "BINARY"
            }
        ]
    }
}
```

Write the Dataloader

The second step is to write the *dataloader*. The function of *dataloader* is to read the dataset into a `Dataset` object. The *code block* below displays the "LeedsSportsPose" dataloader.

```python
#!/usr/bin/env python3
#
# Copyright 2021 Graviti. Licensed under MIT License.
#
# pylint: disable=invalid-name

"""Dataloader of the LeedsSportsPose dataset."""

import os

from ...dataset import Data, Dataset
from ...geometry import Keypoint2D
from ...label import LabeledKeypoints2D
from ..._utility import glob

DATASET_NAME = "LeedsSportsPose"


def LeedsSportsPose(path: str) -> Dataset:
    """Dataloader of the LeedsSportsPose dataset.

    Arguments:
        path: The root directory of the dataset.
            The folder structure should be like::

                <path>
                    joints.mat
                    images/
                        im0001.jpg
                        im0002.jpg
                        ...

    Returns:
        Loaded `Dataset` object.

    """
    from scipy.io import loadmat  # pylint: disable=import-outside-toplevel

    root_path = os.path.abspath(os.path.expanduser(path))

    dataset = Dataset(DATASET_NAME)
    dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))
    segment = dataset.create_segment()

    mat = loadmat(os.path.join(root_path, "joints.mat"))

    joints = mat["joints"].T
    image_paths = glob(os.path.join(root_path, "images", "*.jpg"))
    for image_path in image_paths:
        data = Data(image_path)
        data.label.keypoints2d = []
        index = int(os.path.basename(image_path)[2:6]) - 1  # get image index from
↪"im0001.jpg"
```

(continues on next page)

(continued from previous page)

```
53
54          keypoints = LabeledKeypoints2D()
55          for keypoint in joints[index]:
56              keypoints.append(  # pylint: disable=no-member  # pylint issue #3131
57                  Keypoint2D(keypoint[0], keypoint[1], int(not keypoint[2]))
58              )
59
60          data.label.keypoints2d.append(keypoints)
61          segment.append(data)
62      return dataset
```

Note that after creating the *dataset*, you need to load the *catalog*.(L42) The catalog file "catalog.json" is in the same directory with dataloader file.

In this example, we create a default segment without giving a specific name. You can also create a segment by `dataset.create_segment(SEGMENT_NAME)`.

See *this page* for more details for about Keypoints2D annotation details.

Note: The *LeedsSportsPose dataloader* above uses relative import(L11-13). However, when you write your own dataloader you should use regular import. And when you want to contribute your own dataloader, remember to use relative import.

Upload Dataset

After you finish the *dataloader* and organize the "LeedsSportsPose" into a `Dataset` object, you can upload it to TensorBay for sharing, reuse, etc.

```
# dataset is the one you initialized in "Organize Dataset" section
dataset_client = gas.upload_dataset(dataset, jobs=8, skip_uploaded_files=False)
dataset_client.commit("LeedsSportsPose")
```

Remember to execute the commit step after uploading. If needed, you can re-upload and commit again. Please see *this page* for more details about version control.

Note: Commit operation can also be done on our GAS Platform.

Read Dataset

Now you can read "LeedsSportsPose" dataset from TensorBay.

```
dataset_client = gas.get_dataset("LeedsSportsPose")
```

In *dataset* "LeedsSportsPose", there is one default *Segments* `""` (empty string). You can get it by passing the segment name.

```
from tensorbay.dataset import Segment

default_segment = Segment("", dataset_client)
```

In the train *segment*, there is a sequence of *data*. You can get one by index.

```
data = default_segment[0]
```

Note: If the *segment* or advanced_features/fusion_dataset/fusion_dataset_structure:fusion segment is created without given name, then its name will be "".

In each *data*, there is a sequence of *Keypoints2D* annotations. You can get one by index.

```
label_keypoints2d = data.label.keypoints2d[0]
x = data.label.keypoints2d[0][0].x
y = data.label.keypoints2d[0][0].y
v = data.label.keypoints2d[0][0].v
```

There is only one label type in "LeedsSportsPose" dataset, which is `keypoints2d`. The information stored in `x` (`y`) is the x (y) coordinate of one keypoint of one keypoints list. The information stored in `v` is the visible status of one keypoint of one keypoints list. See *this page* for more details about the structure of Keypoints2D.

Delete Dataset

To delete "LeedsSportsPose", run the following code:

```
gas.delete_dataset("LeedsSportsPose")
```

1.2.4 Neolix OD

This topic describes how to manage the "Neolix OD" dataset.

"Neolix OD" is a dataset with *Box3D* label type (Fig. 1.3). See this page for more details about this dataset.

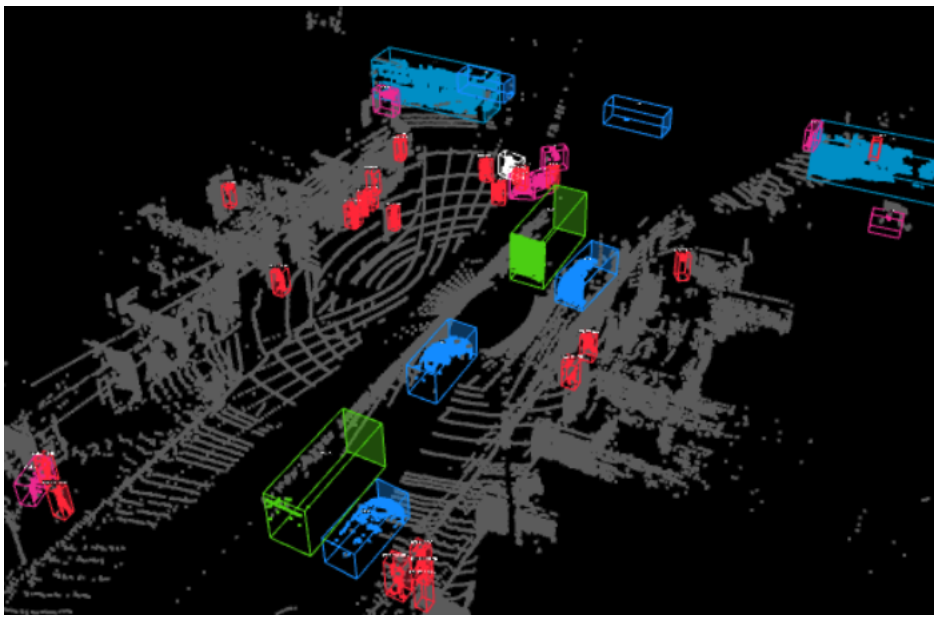

Fig. 1.3: The preview of a point cloud from "Neolix OD" with Box3D labels.

Authorize a Client Object

First of all, create a GAS client.

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
```

Create Dataset

Then, create a dataset client by passing the dataset name to the GAS client.

```
gas.create_dataset("Neolix OD")
```

List Dataset Names

To check if you have created "Neolix OD" dataset, you can list all your available datasets. See *this page* for details.

```
list(gas.list_dataset_names())
```

Note: Note that method `list_dataset_names()` returns an iterator, use `list()` to transfer it to a "list".

Organize Dataset

Now we describe how to organize the "Neolix OD" dataset by the *`Dataset`* object before uploading it to TensorBay. It takes the following steps to organize "Neolix OD".

Write the Catalog

The first step is to write the *catalog*. Catalog is a json file contains all label information of one dataset. See *this page* for more details. The only annotation type for "Neolix OD" is *Box3D*, and there are 15 *Category* types and 3 *Attributes* types.

```
{
    "BOX3D": {
        "categories": [
            { "name": "Adult" },
            { "name": "Animal" },
            { "name": "Barrier" },
            { "name": "Bicycle" },
            { "name": "Bicycles" },
            { "name": "Bus" },
            { "name": "Car" },
            { "name": "Child" },
            { "name": "Cyclist" },
            { "name": "Motorcycle" },
            { "name": "Motorcyclist" },
            { "name": "Trailer" },
            { "name": "Tricycle" },
```

(continues on next page)

(continued from previous page)

```
            { "name": "Truck" },
            { "name": "Unknown" }
        ],
        "attributes": [
            {
                "name": "Alpha",
                "type": "number",
                "description": "Angle of view"
            },
            {
                "name": "Occlusion",
                "enum": [0, 1, 2],
                "description": "It indicates the degree of occlusion of objects by other obstacles"
            },
            {
                "name": "Truncation",
                "type": "boolean",
                "description": "It indicates whether the object is truncated by the edge of the image"
            }
        ]
    }
}
```

Write the Dataloader

The second step is to write the *dataloader*. The function of *dataloader* is to read the dataset into a `Dataset` object. The *code block* below displays the "Neolix OD" dataloader.

```
#!/usr/bin/env python3
#
# Copyright 2021 Graviti. Licensed under MIT License.
#
# pylint: disable=invalid-name

"""Dataloader of the NeolixOD dataset."""

import os

from quaternion import from_rotation_vector

from ...dataset import Data, Dataset
from ...label import LabeledBox3D
from ..._utility import glob

DATASET_NAME = "NeolixOD"


def NeolixOD(path: str) -> Dataset:
    """Dataloader of the NeolixOD dataset.

    Arguments:
        path: The root directory of the dataset.
            The file structure should be like::
```

(continues on next page)

(continued from previous page)

```
                <path>
                    bins/
                        <id>.bin
                    labels/
                        <id>.txt
                    ...

    Returns:
        Loaded `Dataset` object.

    """
    root_path = os.path.abspath(os.path.expanduser(path))

    dataset = Dataset(DATASET_NAME)
    dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))
    segment = dataset.create_segment()

    point_cloud_paths = glob(os.path.join(root_path, "bins", "*.bin"))

    for point_cloud_path in point_cloud_paths:
        data = Data(point_cloud_path)
        data.label.box3d = []

        point_cloud_id = os.path.basename(point_cloud_path)[:6]
        label_path = os.path.join(root_path, "labels", f"{point_cloud_id}.txt")

        with open(label_path, encoding="utf-8") as fp:
            for label_value_raw in fp:
                label_value = label_value_raw.rstrip().split()
                label = LabeledBox3D(
                    category=label_value[0],
                    attributes={
                        "Occlusion": int(label_value[1]),
                        "Truncation": bool(int(label_value[2])),
                        "Alpha": float(label_value[3]),
                    },
                    size=[float(label_value[10]), float(label_value[9]), float(label_
↪value[8])],
                    translation=[
                        float(label_value[11]),
                        float(label_value[12]),
                        float(label_value[13]) + 0.5 * float(label_value[8]),
                    ],
                    rotation=from_rotation_vector((0, 0, float(label_value[14]))),
                )
                data.label.box3d.append(label)

        segment.append(data)
    return dataset
```

Note that after creating the *dataset*, you need to load the *catalog*.(L39) The catalog file "catalog.json" is in the same directory with dataloader file.

In this example, we create segments by `dataset.create_segment(SEGMENT_NAME)`. You can also create a default segment without giving a specific name, then its name will be "".

See *this page* for more details for about Box3D annotation details.

Note: The *Neolix OD dataloader* above uses relative import(L11-12). However, when you write your own dataloader you should use regular import. And when you want to contribute your own dataloader, remember to use relative import.

Upload Dataset

After you finish the *dataloader* and organize the "Neolix OD" into a `Dataset` object, you can upload it to TensorBay for sharing, reuse, etc.

```
# dataset is the one you initialized in "Organize Dataset" section
dataset_client = gas.upload_dataset(dataset, jobs=8, skip_uploaded_files=False)
dataset_client.commit("Neolix OD")
```

Remember to execute the commit step after uploading. If needed, you can re-upload and commit again. Please see *this page* for more details about version control.

Note: Commit operation can also be done on our GAS Platform.

Read Dataset

Now you can read "Neolix OD" dataset from TensorBay.

```
dataset_client = gas.get_dataset("Neolix OD")
```

In *dataset* "Neolix OD", there is one default *Segment*: `""` (empty string). You can get a segment by passing the required segment name.

```
from tensorbay.dataset import Segment

default_segment = Segment("", dataset_client)
```

In the default *segment*, there is a sequence of *data*. You can get one by index.

```
data = default_segment[0]
```

Note: If the *segment* or advanced_features/fusion_dataset/fusion_dataset_structure:fusion segment is created without given name, then its name will be "".

In each *data*, there is a sequence of *Box3D* annotations. You can get one by index.

```
label_box3d = data.label.box3d[0]
category = label_box3d.category
attributes = label_box3d.attributes
```

There is only one label type in "Neolix OD" dataset, which is `box3d`. The information stored in *Category* is one of the category names in "categories" list of *catalog.json*. The information stored in *Attributes* is one of the attributes in "attributes" list of *catalog.json*.

See *this page* for more details about the structure of Box3D.

Delete Dataset

To delete "Neolix OD", run the following code:

```
gas.delete_dataset("Neolix OD")
```

1.2.5 THCHS-30

This topic describes how to manage the "THCHS-30" dataset.

"THCHS-30" is a dataset with *Sentence* label type. See this page for more details about this dataset.

Authorize a Client Object

First of all, create a GAS client.

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
```

Create Dataset

Then, create a dataset client by passing the dataset name to the GAS client.

```
gas.create_dataset("THCHS-30")
```

List Dataset Names

To check if you have created "THCHS-30" dataset, you can list all your available datasets. See *this page* for details.

```
list(gas.list_dataset_names())
```

Note: Note that method `list_dataset_names()` returns an iterator, use `list()` to transfer it to a "list".

Organize Dataset

Now we describe how to organize the "THCHS-30" dataset by the *`Dataset`* object before uploading it to TensorBay. It takes the following steps to organize "THCHS-30".

Write the Catalog

The first step is to write the *catalog*. Typically, Catalog is a json file contains all label information of one dataset. See *this page* for more details. However the catalog of `THCHS-30` is too large, so we need to load the subcatalog by the raw file and map it to catalog, See *code block* below for more details.

Write the Dataloader

The second step is to write the *dataloader*. The function of *dataloader* is to read the dataset into a `Dataset` object. The *code block* below displays the "THCHS-30" dataloader.

```python
#!/usr/bin/env python3
#
# Copyright 2021 Graviti. Licensed under MIT License.
#
# pylint: disable=invalid-name

"""Dataloader of the THCHS-30 dataset."""

import os
from itertools import islice
from typing import List

from ...dataset import Data, Dataset
from ...label import LabeledSentence, SentenceSubcatalog, Word
from ..._utility import glob

DATASET_NAME = "THCHS-30"
_SEGMENT_NAME_LIST = ("train", "dev", "test")


def THCHS30(path: str) -> Dataset:
    """Dataloader of the THCHS-30 dataset.

    Arguments:
        path: The root directory of the dataset.
            The file structure should be like::

                <path>
                    lm_word/
                        lexicon.txt
                    data/
                        A11_0.wav.trn
                        ...
                    dev/
                        A11_101.wav
                        ...
                    train/
                    test/

    Returns:
        Loaded `Dataset` object.

    """
    dataset = Dataset(DATASET_NAME)
    dataset.catalog.sentence = _get_subcatalog(os.path.join(path, "lm_word", "lexicon.
↪txt"))
```

(continues on next page)

(continued from previous page)

```
    for segment_name in _SEGMENT_NAME_LIST:
        segment = dataset.create_segment(segment_name)
        for filename in glob(os.path.join(path, segment_name, "*.wav")):
            data = Data(filename)
            label_file = os.path.join(path, "data", os.path.basename(filename) + ".trn
↪")
            data.label.sentence = _get_label(label_file)
            segment.append(data)
    return dataset


def _get_label(label_file: str) -> List[LabeledSentence]:
    with open(label_file, encoding="utf-8") as fp:
        labels = ((Word(text=text) for text in texts.split()) for texts in fp)
        return [LabeledSentence(*labels)]


def _get_subcatalog(lexion_path: str) -> SentenceSubcatalog:
    subcatalog = SentenceSubcatalog()
    with open(lexion_path, encoding="utf-8") as fp:
        for line in islice(fp, 4, None):
            subcatalog.append_lexicon(line.strip().split())
    return subcatalog
```

Normally, after creating the *dataset*, you need to load the *catalog*. However, in this example, there is no `catalog.json` file, because the lexion of `THCHS-30` is too large (See more details of lexion in *Sentence*). Therefore, We load subcatalog from the raw file lexicon.txt and map it to have the catalog.(L45)

See *this page* for more details about Sentence annotation details.

Note: The *THCHS-30 dataloader* above uses relative import(L13-14). However, when you write your own dataloader you should use regular import. And when you want to contribute your own dataloader, remember to use relative import.

Upload Dataset

After you finish the *dataloader* and organize the "THCHS-30" into a `Dataset` object, you can upload it to TensorBay for sharing, reuse, etc.

```
# dataset is the one you initialized in "Organize Dataset" section
dataset_client = gas.upload_dataset(dataset, jobs=8, skip_uploaded_files=False)
dataset_client.commit("THCHS-30")
```

Remember to execute the commit step after uploading. If needed, you can re-upload and commit again. Please see *Version Control* for more details.

Note:

Commit operation can alse be done on our GAS Platform.

Read Dataset

Now you can read "THCHS-30" dataset from TensorBay.

```
dataset_client = gas.get_dataset("THCHS-30")
```

In *dataset* "THCHS-30", there are three *Segments*: `dev`, `train` and `test`, you can get the segment names by list them all.

```
list(dataset_client.list_segment_names())
```

You can get a segment by passing the required segment name.

```
from tensorbay.dataset import Segment

dev_segment = Segment("dev", dataset_client)
```

In the dev *segment*, there is a sequence of *data*. You can get one by index.

```
data = dev_segment[0]
```

Note: If the *segment* or advanced_features/fusion_dataset/fusion_dataset_structure:fusion segment is created without given name, then its name will be "".

In each *data*, there is a sequence of *Sentence* annotations. You can get one by index.

```
labeled_sentence = data.label.sentence[0]
sentence = labeled_sentence.sentence
spell = labeled_sentence.spell
phone = labeled_sentence.phone
```

There is only one label type in "THCHS-30" dataset, which is `Sentence`. It contains `sentence`, `spell` and `phone` information. See *this page* for more details about the structure of Sentence.

Delete Dataset

To delete "THCHS-30", run the following code:

```
gas.delete_dataset("THCHS-30")
```

1.2.6 Read "Dataset" Class

This topic describes how to read the `Dataset` class after you have *organized the "BSTLD" dataset*. See this page for more details about this dataset.

As mentioned in *Dataset Management*, you need to write a *dataloader* to get a `Dataset`. However, there are already a number of dataloaders in TensorBay SDK provided by the community. Thus, instead of writing, you can just import an available dataloader.

The local directory structure for "BSTLD" should be like:

```
<path>
    rgb/
        additional/
            2015-10-05-10-52-01_bag/
                <image_name>.jpg
                ...
            ...
        test/
            <image_name>.jpg
            ...
        train/
            2015-05-29-15-29-39_arastradero_traffic_light_loop_bag/
                <image_name>.jpg
                ...
            ...
    test.yaml
    train.yaml
    additional_train.yaml
```

```
from tensorbay.opendataset import BSTLD

dataset = BSTLD("path/to/dataset/directory")
```

Warning: Dataloaders provided by the community work well only with the original dataset directory structure. Downloading datasets from either official website or Graviti Opendatset Platform is highly recommended.

TensorBay supplies two methods to fetch *segment* from *dataset*.

```
train_segment = dataset.get_segment_by_name("train")
first_segment = dataset[0]
```

The *segment* you get now is the same as the one you *read from TensorBay*. In the train *segment*, there is a sequence of *data*. You can get one by index.

```
data = train_segment[3]
```

In each *data*, there is a sequence of *Box2D* annotations. You can get one by index.

```
label_box2d = data.label.box2d[0]
category = label_box2d.category
attributes = label_box2d.attributes
```

1.3 Dataset Management

This topic describes the key operations towards your datasets, including:

- *Organize Dataset*
- *Upload Dataset*
- *Read Dataset*

1.3.1 Organize Dataset

TensorBay SDK supports methods to organize your local datasets into uniform TensorBay dataset strucutre (*ref*). The typical steps to organize a local dataset:

- First, write a dataloader (*ref*) to load the whole local dataset into a `Dataset` instance,
- Second, write a catalog (*ref*) to store all the label meta information inside a dataset.

Note: A catalog is needed only if there is label information inside the dataset.

This part is an example for organizing a dataset.

1.3.2 Upload Dataset

There are two usages for the organized local dataset (i.e. the initialized `Dataset` instance):

- Upload it to TensorBay.
- Use it directly.

In this section, we mainly discuss the uploading operation. See *this example* for details about the latter usage.

There are plenty of benefits of uploading local datasets to TensorBay.

- Reuse: you can reuse your datasets without preprocessing again.
- Share: you can share them with your team or the community.
- Preview: you can preview your datasets without coding.
- Version control: you can upload different versions of one dataset and control them conveniently.

This part is an example for uploading a dataset.

1.3.3 Read Dataset

There are two types of datasets you can read from TensorBay:

- Datasets uploaded by yourself as mentioned in *Upload Dataset*.
- Datasets uplaoded by the community (i.e. the open datasets).

Note: Before reading a dataset uploaded by the community, you need to fork it first.

Note: You can visit our Graviti AI Service(GAS) platform to check the dataset details, such as dataset name, version information, etc.

This part is an example for reading a dataset.

1.4 Version Control

TensorBay currently supports the linear version control. A new version of a dataset can be built upon the previous version. Figure. 1.4 demonstrates the relations between different versions of a dataset.

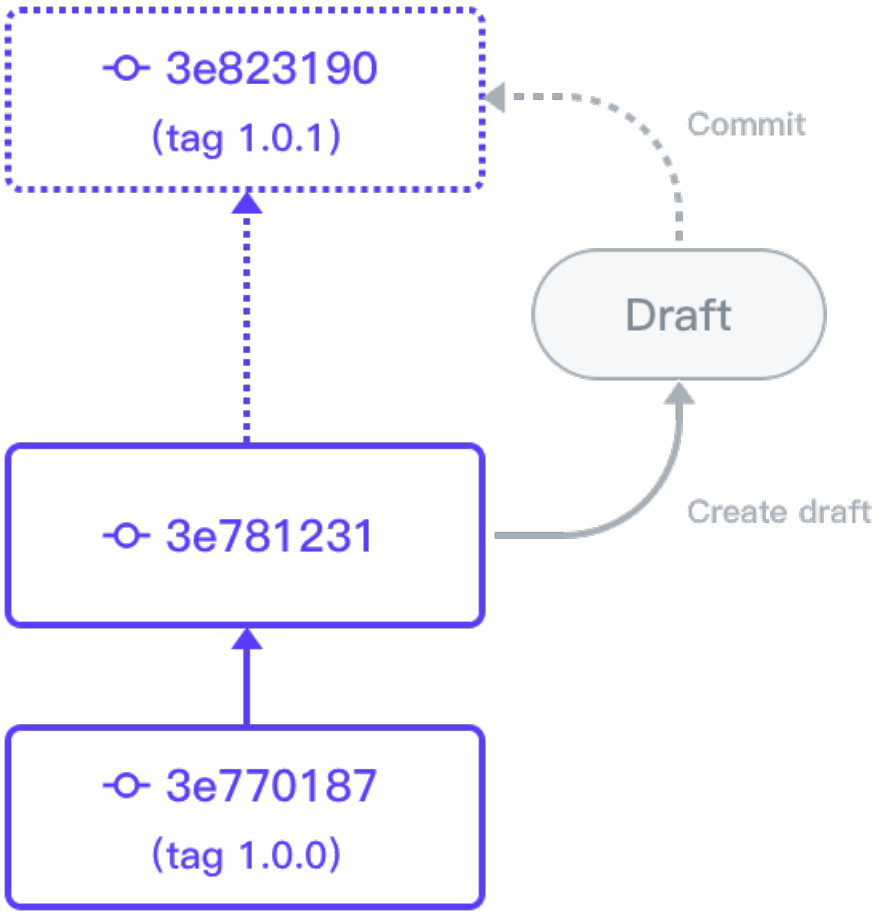

Fig. 1.4: The relations between different versions of a dataset.

1.4.1 Create Draft And Commit

The version control is based on the *draft* and *commit*.

The `GAS` is actually responsible for operating the datasets, while the `DatasetClient` is for operating content of one dataset in the draft or commit. A certain client can only work in the draft or commit. Also, the dataset client supports the function of version control. You can create and close the draft with the given methods in the dataset client.

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
```

(continues on next page)

(continued from previous page)

```
# dataset is the original dataset.
# actually when calling "gas.upload_dataset(dataset.name)", a default draft "" is␣
↪created.
gas.create_dataset(dataset.name)
dataset_client = gas.upload_dataset(dataset)
dataset_client.commit("first_commit")

# segment contains extra data that you want to add to the dataset.
# create the draft.
dataset_client.create_draft("draft-2")
dataset_client.upload_segment(segment)
# commit the draft and the draft will be deleted.
dataset_client.commit("second_commit")
```

1.4.2 Checkout

You can checkout to other draft with draft number or to commit with commit id through *checkout()* in the *DatasetClient*. The draft number can be found through *list_draft_titles_and_numbers()* and the commit id can be can be found through the web page.

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)

dataset_client = gas.create_dataset(dataset.name)

dataset_client.create_draft("draft-1")
dataset_client.commit("first_commit")

dataset_client.create_draft("draft-2")
dataset_client.commit("second_commit")

dataset_client.create_draft("draft-3")
# list draft numbers.
drafts = list(dataset_client.list_draft_titles_and_numbers())

# checkout to the draft.
dataset_client.checkout(draft_number=draft_number)
# checkout to the commit.
dataset_client.checkout(commit=commit_id)
```

1.5 Getting Started with CLI

The TensorBay Command Line Interface is a tool to operate on your datasets. It supports Windows, Linux, and Mac platforms.

You can use TensorBay CLI to:

- Create and delete dataset.
- List data, segments and datasets on TensorBay.

- Upload data to TensorBay.

1.5.1 Installation

To use TensorBay CLI, please install TensorBay SDK first.

```
$ pip3 install tensorbay
```

1.5.2 TBRN

TensorBay Resource Name(TBRN) uniquely defines the data stored in TensorBay. TBRN begins with `tb:`. Default segment can be defined as `""` (empty string). The following is the general format for TBRN:

```
tb:[dataset_name]:[segment_name]://[remote_path]
```

1.5.3 Configuration

Use the command below to configure the accessKey and URL(optional).

```
$ gas config [accessKey] [url]
```

AccessKey is used for identification when using TensorBay to operate on your dataset. The default url is "https://gas.graviti.cn/".

You can set the accessKey and URL into configuration:

```
$ gas config Accesskey-***** https://gas.graviti.cn/
```

To show configuration information:

```
$ gas config
```

1.6 Dataset Management

TensorBay CLI offers following sub-commands to manage your dataset. (Table. 1.2)

Table 1.2: Sub-Commands

Sub-Commands	Description
create	Create a dataset
ls	List data, segments and datasets
delete	Delete a dataset

1.6.1 Create dataset

The basic structure of the sub-command to create a dataset with given name:

```
$ gas create [tbrn]

tbrn:
    tb:[dataset_name]
```

Take BSTLD for example:

```
$ gas create tb:BSTLD
```

1.6.2 Read Dataset

The basic structure of the sub-command to List data, segments and datasets:

```
$ gas ls [Options] [tbrn]

Options:
  -a, --all     List all files under all segments.
                 Only works when [tbrn] is tb:[dataset_name].

tbrn:
  None
  tb:[dataset_name]
  tb:[dataset_name]:[segment_name]
  tb:[dataset_name]:[segment_name]://[remote_path]
```

If the path is empty, list the names of all datasets. You can list data in the following ways:

1. List the names of all datasets.

```
$ gas ls
```

2. List the names of all segments of BSTLD.

```
$ gas ls tb:BSTLD
```

3. List all the files in all the segments of BSTLD.

```
$ gas ls -a tb:BSTLD
```

4. List all the files in the `train` segment of BSTLD.

```
$ gas ls tb:BSTLD:train
```

1.6.3 Delete Dataset

The basic structure of the sub-command to delete the dataset with given name:

```
$ gas delete [tbrn]

tbrn:
  tb:[dataset_name]
```

Take BSTLD for example:

```
$ gas delete tb:BSTLD
```

1.7 Glossary

1.7.1 accesskey

An accesskey is an access credential for identification when using TensorBay to operate on your dataset.

To obtain an accesskey, you need to log in to Graviti AI Service(GAS) and visit the developer page to create one.

For the usage of accesskey via Tensorbay SDK or CLI, please see *SDK authorization* or *CLI configration*.

1.7.2 dataset

A uniform dataset format defined by Tensorbay, which only contains one type of data collected from one sensor or without sensor information.

The corresponding class of dataset is `Dataset`.

See *Dataset Structure* for more details.

1.7.3 dataloader

A function that can organize files within a formatted folder into a `Dataset` instance

The only input of the function should be a str indicating the path to the folder containing the dataset, and the return value should be the loaded `Dataset`

```
from tensorbay.dataset import Dataset


def DatasetName(path: str) -> Dataset:
    """The dataloader of <Dataset Name>  dataset.

    Arguments:
        path: The root directory of the dataset.
            The file structure should be like::

```

(continues on next page)

(continued from previous page)

```
                <path>
                    structure under the path

    Returns:
        The loaded 'Dataset' object.

    """
    dataset = Dataset("<Dataset Name>")
    ...  # organize the files( and the labels) under the path to the dataset
    return dataset
```

Note: The name of the dataloader function is a unique indentification of the dataset. It is in upper camel case and is generally obtained by removing special characters from the dataset name.

Take Dogs vs Cats dataset as an example, the name of its dataloader function is `DogsVsCats()`.

See more dataloader examples in *tensorbay.opendataset*.

1.7.4 TBRN

TBRN is the abbreviation for TensorBay Resource Name, which represents the data or a collection of data stored in TensorBay uniquely.

Note that TBRN is only used in *CLI*.

TBRN begins with `tb:`, followed by the dataset name, the segment name and the file name.

The following is the general format for TBRN:

```
tb:[dataset_name]:[segment_name]://[remote_path]
```

Suppose we have an image `000000.jpg` under the default segment of a dataset named `example`, then we have the TBRN of this image:

```
tb:example::/000000.jpg
```

Note: Default segment is defined as `""` (empty string).

1.7.5 commit

Similar with Git, a commit is a version of a dataset, which contains the changes compared with the former commit. You can view a certain commit of a dataset based on the given commit ID.

A commit is readable, but is not writable. Thus, only read operations such as getting catalog, files and labels are allowed. To change a dataset, please create a new commit. See *draft* for details.

On the other hand, "commit" also represents the action to save the changes inside a *draft* into a commit.

1.7.6 draft

Similar with Git, a draft is a workspace in which changing the dataset is allowed.

A draft is created based on a *commit*, and the changes inside it will be made into a commit.

There are scenarios when modifications of a dataset are required, such as correcting errors, enlarging dataset, adding more types of labels, etc. Under these circumstances, you can create a draft, edit the dataset and commit the draft.

1.8 Dataset Structure

For ease of use, TensorBay defines a uniform dataset format. In this topic, we explain the related concepts. The TensorBay dataset format looks like:

```
dataset
├── catalog
│   ├── subcatalog
│   ├── subcatalog
│   └── ...
├── segment
│   ├── data
│   ├── data
│   └── ...
├── segment
│   ├── data
│   ├── data
│   └── ...
└── ...
```

1.8.1 dataset

Dataset is the topmost concept in TensorBay dataset format. Each dataset includes a catalog and a certain number of segments.

The corresponding class of dataset is `Dataset`.

1.8.2 catalog

Catalog is used for storing label meta information. It collects all the labels corresponding to a dataset. There could be one or several subcatalogs (*Label Format*) under one catalog, each of which only stores label meta information of one label type.

For example, there is only one subcatalog ("BOX3D") in the catalog of dataset Neolix OD.

```
{
    "BOX3D": {
        "categories": [
            { "name": "Adult" },
            { "name": "Animal" },
            { "name": "Barrier" },
            { "name": "Bicycle" },
            { "name": "Bicycles" },
            { "name": "Bus" },
            { "name": "Car" },
```

(continues on next page)

(continued from previous page)

```
            { "name": "Child" },
            { "name": "Cyclist" },
            { "name": "Motorcycle" },
            { "name": "Motorcyclist" },
            { "name": "Trailer" },
            { "name": "Tricycle" },
            { "name": "Truck" },
            { "name": "Unknown" }
        ],
        "attributes": [
            {
                "name": "Alpha",
                "type": "number",
                "description": "Angle of view"
            },
            {
                "name": "Occlusion",
                "enum": [0, 1, 2],
                "description": "It indicates the degree of occlusion of objects by␣
↪other obstacles"
            },
            {
                "name": "Truncation",
                "type": "boolean",
                "description": "It indicates whether the object is truncated by the␣
↪edge of the image"
            }
        ]
    }
}
```

Note that catalog is not needed if there is no label information in a dataset.

1.8.3 segment

There may be several parts in a dataset. In TensorBay format, each part of the dataset is stored in one segment. For example, all training samples of a dataset can be organized in a segment named "train".

The corresponding class of segment is *Segment*.

1.8.4 data

Data is the structural level next to segment. One data contains one dataset sample and its related labels, as well as any other information such as timestamp.

The corresponding class of data is *Data*.

1.9 Label Format

TensorBay supports multiple types of labels.

Each *Data* object can have multiple types of `label`.

And each type of `label` is supported with a specific label class, and has a corresponding *subcatalog* class.

Table 1.3: supported label types

supported label types	label classes	subcatalog classes
Classification	*Classification*	*ClassificationSubcatalog*
Box2D	*LabeledBox2D*	*Box2DSubcatalog*
Box3D	*LabeledBox3D*	*Box3DSubcatalog*
Keypoints2D	*LabeledKeypoints2D*	*Keypoints2DSubcatalog*
Sentence	*LabeledSentence*	SetenceSubcatalog

1.9.1 Common Label Properties

Different types of labels contain differenct aspects of annotation information about the data. Some are more general, and some are unique to a specific label type.

We first introduce three common properties of a label, and the unique ones will be explained under the corresponding type of label.

Here we take a *2D box label* as an example:

```
>>> from tensorbay.label import LabeledBox2D
>>> label = LabeledBox2D(
... 10, 20, 30, 40,
... category="category",
... attributes={"attribute_name": "attribute_value"},
... instance="instance_ID"
... )
```

Category

Category is a string indicating the class of the labeled object.

```
>>> label.category
'data_category'
```

Attributes

Attributes are the additional information about this data, and there is no limit on the number of attributes.

The attribute names and values are stored in key-value pairs.

```
>>> label.attributes
{'attribute_name': 'attribute_value'}
```

Instance

Instance is the unique id for the object inside of the label, which is mostly used for tracking tasks.

```
>>> label.instance
"instance_ID"
```

1.9.2 Common Subcatalog Properties

Before creating a label or adding a label to data, you need to define the annotation rules of the specific label type inside the dataset, which is subcatalog.

Different label types have different subcatalog classes.

Here we take *Box2DSubcatalog* as an example to describe some common features of subcatalog.

```
>>> from tensorbay.label import Box2DSubcatalog
>>> box2d_subcatalog = Box2DSubcatalog(is_tracking=True)
```

TrackingInformation

If the label of this type in the dataset has the information of instance IDs, then the subcatalog should set a flag to show its support for tracking information.

You can pass `True` to the `is_tracking` parameter while creating the subcatalog, or you can set the `is_tracking` attr after initialization.

```
>>> box2d_subcatalog.is_tracking = True
```

CategoryInformation

If the label of this type in the dataset has category, then the subcatalog should contain all the optional categories.

Each *category* of a label appeared in the dataset should be within the categories of the subcatalog.

You can add category information to the subcatalog.

```
>>> box2d_subcatalog.add_category(name="cat", description="The Flerken")
>>> box2d_subcatalog.categories
NameOrderedDict {
  'cat': CategoryInfo("cat")
}
```

We use *CategoryInfo* to describe a *category*. See details in *CategoryInfo*.

AttributesInformation

If the label of this type in the dataset has attributes, then the subcatalog should contain all the rules for different attributes.

Each *attribute* of a label appeared in the dataset should follow the rules set in the attributes of the subcatalog.

You can add attribute information to the subcatalog.

```
>>> box2d_subcatalog.add_attribute(
... name="attribute_name",
... type_="number",
... maximum=100,
... minimum=0,
... description="attribute description"
... )
>>> box2d_subcatalog.categories
NameOrderedDict {
  'attribute_name': AttributeInfo("attribute_name")(...)
}
```

We use *AttributeInfo* to describe the rules of an *attribute*, which refers to the Json schema method.

See details in *AttributeInfo*.

Other unique subcatalog features will be explained in the corresponding label type section.

1.9.3 Classification

Classification is to classify data into different categories.

It is the annotation for the entire file, so each data can only be assigned with one classification label.

Classification labels applies to different types of data, such as images and texts.

The structure of one classification label is like:

```
{
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    }
}
```

To create a *Classification* label:

```
>>> from tensorbay.label import Classification
>>> classification_label = Classification(
... category="data_category",
... attributes={"attribute_name": "attribute_value"}
... )
>>> classification_label
Classification(
  (category): 'data_category',
  (attributes): {...}
)
```

Classification.Category

The category of the entire data file. See *Category* for details.

Classification.Attributes

The attributes of the entire data file. See *Attributes* for details.

Note: There must be either a category or attributes in one classification label.

ClassificationSubcatalog

Before adding the classification label to data, `ClassificationSubcatalog` should be defined.

`ClassificationSubcatalog` has categories and attributes information, see *CategoryInformation* and *AttributesInformation* for details.

To add a `Classification` label to one data:

```
>>> from tensorbay.dataset import Data
>>> data = Data("local_path")
>>> data.label.classification = classification_label
```

Note: One data can only have one classification label.

1.9.4 Box2D

Box2D is a type of label with a 2D bounding box on an image. It's usually used for object detection task.

Each data can be assigned with multiple Box2D label.

The structure of one Box2D label is like:

```
{
    "box2d": {
        "xmin": <float>
        "ymin": <float>
        "xmax": <float>
        "ymax": <float>
    },
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

To create a `LabeledBox2D` label:

```
>>> from tensorbay.label import LabeledBox2D
>>> box2d_label = LabeledBox2D(
... xmin, ymin, xmax, ymax,
... category="category",
... attributes={"attribute_name": "attribute_value"},
... instance="instance_ID"
... )
>>> box2d_label
LabeledBox2D(xmin, ymin, xmax, ymax)(
  (category): 'category',
  (attributes): {...}
  (instance): 'instance_ID'
)
```

Box2D.box2d

LabeledBox2D extends *Box2D*.

To construct a *LabeledBox2D* instance with only the geometry information, you can use the coordinates of the top-left and bottom-right vertexes of the 2D bounding box, or you can use the coordinate of the top-left vertex, the height and the width of the bounding box.

```
>>> LabeledBox2D(10, 20, 30, 40)
LabeledBox2D(10, 20, 30, 40)()
>>> LabeledBox2D(x=10, y=20, width=20, height=20)
LabeledBox2D(10, 20, 30, 40)()
```

It contains the basic geometry information of the 2D bounding box.

```
>>> box2d_label.xmin
10
>>> box2d_label.ymin
20
>>> box2d_label.xmax
30
>>> box2d_label.ymax
40
>>> box2d_label.br
Vector2D(30, 40)
>>> box2d_label.tl
Vector2D(10, 20)
>>> box2d_label.area()
400
```

Box2D.Category

The category of the object inside the 2D bounding box. See *Category* for details.

Box2D.Attributes

Attributes are the additional information about this object, which are stored in key-value pairs. See *Attributes* for details.

Box2D.Instance

Instance is the unique ID for the object inside of the 2D bounding box, which is mostly used for tracking tasks. See *Instance* for details.

Box2DSubcatalog

Before adding the Box2D labels to data, *Box2DSubcatalog* should be defined.

Box2DSubcatalog has categories, attributes and tracking information, see *CategoryInformation*, *AttributesInformation* and *TrackingInformation* for details.

To add a *LabeledBox2D* label to one data:

```
>>> from tensorbay.dataset import Data
>>> data = Data("local_path")
>>> data.label.box2d = []
>>> data.label.box2d.append(box2d_label)
```

Note: One data may contain multiple Box2D labels, so the `Data.label.box2d` must be a list.

1.9.5 Box3D

Box3D is a type of label with a 3D bounding box on point cloud, which is often used for 3D object detection.

Currently, Box3D labels applies to point data only.

Each point cloud can be assigned with multiple Box3D label.

The structure of one Box3D label is like:

```
{
    "box3d": {
        "translation": {
            "x": <float>
            "y": <float>
            "z": <float>
        },
        "rotation": {
            "w": <float>
            "x": <float>
            "y": <float>
            "z": <float>
        },
        "size": {
            "x": <float>
            "y": <float>
            "z": <float>
        }
```

(continues on next page)

(continued from previous page)

```
    },
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

To create a *LabeledBox3D* label:

```
>>> from tensorbay.label import LabeledBox3D
>>> box3d_label = LabeledBox3D(
... translation=[0, 0, 0],
... rotation=[1, 0, 0, 0],
... size=[10, 20, 30],
... category="category",
... attributes={"attribute_name": "attribute_value"},
... instance="instance_ID"
... )
>>> box3d_label
LabeledBox3D(
  (translation): Vector3D(0, 0, 0),
  (rotation): quaternion(1.0, 0.0, 0.0, 0.0),
  (size): Vector3D(10, 20, 30),
  (category): 'category',
  (attributes): {...},
  (instance): 'instance_ID'
)
```

Box3D.box3d

LabeledBox3D extends *Box3D*.

To construct a *LabeledBox3D* instance with only the geometry information, you can use the transform matrix and the size of the 3D bounding box, or you can use translation and rotation to represent the transform of the 3D bounding box.

```
>>> LabeledBox3D(
... [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0]],
... size=[10, 20, 30],
... )
LabeledBox3D(
  (translation): Vector3D(0, 0, 0),
  (rotation): quaternion(1.0, -0.0, -0.0, -0.0),
  (size): Vector3D(10, 20, 30)
)
>>> LabeledBox3D(
... translation=[0, 0, 0],
... rotation=[1, 0, 0, 0],
... size=[10, 20, 30],
... )
LabeledBox3D(
  (translation): Vector3D(0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
  (rotation): quaternion(1.0, 0.0, 0.0, 0.0),
  (size): Vector3D(10, 20, 30)
)
```

It contains the basic geometry information of the 3D bounding box.

```
>>> box3d_label.transform
Transform3D(
  (translation): Vector3D(0, 0, 0),
  (rotation): quaternion(1.0, 0.0, 0.0, 0.0)
)
>>> box3d_label.translation
Vector3D(0, 0, 0)
>>> box3d_label.rotation
quaternion(1.0, 0.0, 0.0, 0.0)
>>> box3d_label.size
Vector3D(10, 20, 30)
>>> box3d_label.volumn()
6000
```

Box3D.Category

The category of the object inside the 3D bounding box. See *Category* for details.

Box3D.Attributes

Attributes are the additional information about this object, which are stored in key-value pairs. See *Attributes* for details.

Box3D.Instance

Instance is the unique id for the object inside of the 3D bounding box, which is mostly used for tracking tasks. See *Instance* for details.

Box3DSubcatalog

Before adding the Box2D labels to data, `Box2DSubcatalog` should be defined.

`Box2DSubcatalog` has categories, attributes and tracking information, see *CategoryInformation*, *AttributesInformation* and *TrackingInformation* for details.

To add a `LabeledBox3D` label to one data:

```
>>> from tensorbay.dataset import Data
>>> data = Data("local_path")
>>> data.label.box3d = []
>>> data.label.box3d.append(box3d_label)
```

Note: One data may contain multiple Box3D labels, so the `Data.label.box3d` must be a list.

1.9.6 Keypoints2D

Keypoints2D is a type of label with a set of 2D keypoints. It is often used for animal and human pose estimation.

Keypoints2D labels mostly applies to images.

Each data can be assigned with multiple Keypoints2D labels.

The structure of one Keypoints2D label is like:

```
{
    "keypoints2d": [
        { "x": <float>
          "y": <float>
          "v": <int>
        },
        ...
        ...
    ],
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

To create a *LabeledKeypoints2D* label:

```
>>> from tensorbay.label import LabeledKeypoints2D
>>> keypoints2d_label = LabeledKeypoints2D(
... [[10, 20], [15, 25], [20, 30]],
... category="category",
... attributes={"attribute_name": "attribute_value"},
... instance="instance_ID"
... )
>>> keypoints2d_label
LabeledKeypoints2D [
  Keypoint2D(10, 20),
  Keypoint2D(15, 25),
  Keypoint2D(20, 30)
](
  (category): 'category',
  (attributes): {...},
  (instance): 'instance_ID'
)
```

Keypoints2D.keypoints2d

LabeledKeypoints2D extends Keypoints2D.

To construct a *LabeledKeypoints2D* instance with only the geometry information, you need the coordinates of the set of 2D keypoints. You can also add the visible status of each 2D keypoint.

```
>>> LabeledKeypoints2D([[10, 20], [15, 25], [20, 30]])
LabeledKeypoints2D [
  Keypoint2D(10, 20),
  Keypoint2D(15, 25),
  Keypoint2D(20, 30)
]()
>>> LabeledKeypoints2D([[10, 20, 0], [15, 25, 1], [20, 30, 1]])
LabeledKeypoints2D [
  Keypoint2D(10, 20, 0),
  Keypoint2D(15, 25, 1),
  Keypoint2D(20, 30, 1)
]()
```

It contains the basic geometry information of the 2D keypoints. And you can access the keypoints by index.

```
>>> keypoints2d_label[0]
Keypoint2D(10, 20)
```

Keypoints2D.Category

The category of the object inside the 3D bounding box. See *Category* for details.

Keypoints2D.Attributes

Attributes are the additional information about this object, which are stored in key-value pairs. See *Attributes* for details.

Keypoints2D.Instance

Instance is the unique ID for the object inside of the 3D bounding box, which is mostly used for tracking tasks. See *Instance* for details.

Keypoints2DSubcatalog

Before adding 2D keypoints labels to the dataset, *Keypoints2DSubcatalog* should be defined.

Besides *AttributesInformation*, *CategoryInformation*, *TrackingInformation* in *Keypoints2DSubcatalog*, it also has *keypoints* to describe a set of keypoints corresponding to certain categories.

```
>>> from tensorbay.label import Keypoints2DSubcatalog
>>> keypoints2d_subcatalog = Keypoints2DSubcatalog()
>>> keypoints2d_subcatalog.add_keypoints(
... 3,
... names=["head", "body", "feet"],
... skeleton=[[0, 1], [1, 2]],
... visible="BINARY",
```

(continues on next page)

(continued from previous page)

```
... parent_categories=["cat"],
... description="keypoints of cats"
... )
>>> keypoints2d_subcatalog.keypoints
[KeypointsInfo(
   (number): 3,
   (names): [...],
   (skeleton): [...],
   (visible): 'BINARY',
   (parent_categories): [...]
 )]
```

We use *KeypointsInfo* to describe a set of 2D keypoints.

The first parameter of *add_keypoints()* is the number of the set of 2D keypoints, which is required.

The `names` is a list of string representing the names for each 2D keypoint, the length of which is consistent with the number.

The `skeleton` is a two-dimensional list indicating the connection between the keypoints.

The `visible` is the visible status that limits the *v* of *Keypoint2D*. It can only be "BINARY" or "TERNARY".

See details in *Keypoint2D*.

The `parent_categories` is a list of categories indicating to which category the keypoints rule applies.

Mostly, `parent_categories` is not given, which means the keypoints rule applies to all the categories of the entire dataset.

To add a *LabeledKeypoints2D* label to one data:

```
>>> from tensorbay.dataset import Data
>>> data = Data("local_path")
>>> data.label.keypoints2d = []
>>> data.label.keypoints2d.append(keypoints2d_label)
```

Note: One data may contain multiple Keypoints2D labels, so the `Data.label.keypoints2d` must be a list.

1.9.7 Sentence

Sentence label is the transcripted sentence of a piece of audio, which is often used for autonomous speech recognition.

Each audio can be assigned with multiple sentence labels.

The structure of one sentence label is like:

```
{
    "sentence": [
        {
            "text":  <str>
            "begin": <float>
            "end":   <float>
        }
        ...
        ...
```

(continues on next page)

(continued from previous page)

```
    ],
    "spell": [
        {
            "text":  <str>
            "begin": <float>
            "end":   <float>
        }
        ...
        ...
    ],
    "phone": [
        {
            "text":  <str>
            "begin": <float>
            "end":   <float>
        }
        ...
        ...
    ],
    "attributes": {
        <key>: <value>,
        ...
        ...
    }
}
```

To create a *LabeledSentence* label:

```
>>> from tensorbay.label import LabeledSentence
>>> from tensorbay.label import Word
>>> sentence_label = LabeledSentence(
... sentence=[Word("text", 1.1, 1.6)],
... spell=[Word("spell", 1.1, 1.6)],
... phone=[Word("phone", 1.1, 1.6)],
... attributes={"attribute_name": "attribute_value"}
... )
>>> sentence_label
LabeledSentence(
  (sentence): [
    Word(
      (text): 'text',
      (begin): 1.1,
      (end): 1.6
    )
  ],
  (spell): [
    Word(
      (text): 'text',
      (begin): 1.1,
      (end): 1.6
    )
  ],
  (phone): [
    Word(
      (text): 'text',
      (begin): 1.1,
      (end): 1.6
```

(continues on next page)

(continued from previous page)

```
    )
  ],
  (attributes): {
    'attribute_name': 'attribute_value'
  }
```

Sentence.sentence

The *sentence* of a *LabeledSentence* is a list of *Word*, representing the transcripted sentence of the audio.

Sentence.spell

The *spell* of a *LabeledSentence* is a list of *Word*, representing the spell within the sentence.

It is only for Chinese language.

Sentence.phone

The *phone* of a *LabeledSentence* is a list of *Word*, representing the phone of the sentence label.

Word

Word is the basic component of a phonetic transcription sentence, containing the content of the word, the start and the end time in the audio.

```
>>> from tensorbay.label import Word
>>> Word("text", 1.1, 1.6)
Word(
  (text): 'text',
  (begin): 1,
  (end): 2
)
```

sentence, *spell*, and *phone* of a sentence label all compose of *Word*.

Sentence.Attributes

The attributes of the transcripted sentence. See *AttributesInformation* for details.

SentenceSubcatalog

Before adding sentence labels to the dataset, `SetenceSubcatalog` should be defined.

Besides *AttributesInformation* in `SetenceSubcatalog`, it also has `is_sample`, `sample_rate` and `lexicon`. to describe the transcripted sentences of the audio.

```
>>> from tensorbay.label import SentenceSubcatalog
>>> sentence_subcatalog = SentenceSubcatalog(
... is_sample=True,
... sample_rate=5,
```

(continues on next page)

(continued from previous page)

```
... lexicon=[["word", "spell", "phone"]]
... )
>>> sentence_subcatalog
SentenceSubcatalog(
  (is_sample): True,
  (sample_rate): 5,
  (lexicon): [...]
)
>>> sentence_subcatalog.lexicon
[['word', 'spell', 'phone']]
```

The `is_sample` is a boolen value indicating whether time format is sample related.

The `sample_rate` is the number of samples of audio carried per second. If `is_sample` is Ture, then `sample_rate` must be provided.

The `lexicon` is a list consists all of text and phone.

Besides giving the parameters while initialing `SetenceSubcatalog`, you can set them after intialization.

```
>>> from tensorbay.label import SentenceSubcatalog
>>> sentence_subcatalog = SentenceSubcatalog()
>>> sentence_subcatalog.is_sample = True
>>> sentence_subcatalog.sample_rate = 5
>>> sentence_subcatalog.append_lexicon(["text", "spell", "phone"])
>>> sentence_subcatalog
SentenceSubcatalog(
  (is_sample): True,
  (sample_rate): 5,
  (lexicon): [...]
)
```

To add a *LabeledSentence* label to one data:

```
>>> from tensorbay.dataset import Data
>>> data = Data("local_path")
>>> data.label.sentence = []
>>> data.label.sentence.append(sentence_label)
```

Note: One data may contain multiple Sentence labels, so the `Data.label.sentence` must be a list.

1.10 API Reference

1.10.1 tensorbay.client

tensorbay.client.cli

Command-line interface.

Use 'gas' + COMMAND in terminal to operate on datasets.

Use 'gas config' to configure environment.

Use 'gas create' to create a dataset.

Use 'gas delete' to delete a dataset.

Use 'gas ls' to list data.

Use 'gas cp' to upload data.

Use 'gas rm' to delete data.

tensorbay.client.dataset

Class DatasetClientBase, DatasetClient and FusionDatasetClient.

DatasetClient is a remote concept. It contains the information needed for determining a unique dataset on TensorBay, and provides a series of methods within dataset scope, such as *DatasetClient.get_segment()*, DatasetClient.list_segment_names(), *DatasetClient.commit*, and so on. In contrast to the *DatasetClient*, *Dataset* is a local concept. It represents a dataset created locally. Please refer to *Dataset* for more information.

Similar to the *DatasetClient*, the FusionDatasetClient represents the fusion dataset on TensorBay, and its local counterpart is FusionDataset. Please refer to FusionDataset for more information.

class tensorbay.client.dataset.**DatasetClient**(*name: str*, *dataset_id: str*, *gas_client: GAS*, ***, *commit_id: Optional[str] = None*)

Bases: *tensorbay.client.dataset.DatasetClientBase*

This class defines *DatasetClient*.

DatasetClient inherits from DataClientBase and provides more methods within a dataset scope, such as *DatasetClient.get_segment()*, *DatasetClient.commit* and *DatasetClient.upload_segment()*. In contrast to FusionDatasetClient, a *DatasetClient* has only one sensor.

get_or_create_segment(*name: str = ''*) → *tensorbay.client.segment.SegmentClient*

Create a segment with the given name to the draft.

Parameters **name** – Segment name, can not be "_default".

Returns Created *SegmentClient* with given name.

get_segment(*name: str = ''*) → *tensorbay.client.segment.SegmentClient*

Get a segment in a certain commit according to given name.

Parameters **name** – The name of the required segment.

Returns *~tensorbay.client.segment.SegmentClient*.

Return type The required class

Raises ***GASSegmentError*** – When the required segment does not exist.

upload_segment(*segment:* tensorbay.dataset.segment.Segment, ***, *jobs: int = 1*, *skip_uploaded_files: bool = False*) → *tensorbay.client.segment.SegmentClient*

Upload a *Segment* to the dataset.

This function will upload all info contains in the input *Segment*, which includes: - Create a segment using the name of input Segment. - Upload all Data in the Segment to the dataset.

Parameters

- **segment** – The *Segment* contains the information needs to be upload.
- **jobs** – The number of the max workers in multi-thread uploading method.
- **skip_uploaded_files** – True for skipping the uploaded files.

Returns

The *SegmentClient* used for uploading the data in the segment.

class tensorbay.client.dataset.**DatasetClientBase**(*name: str*, *dataset_id: str*, *gas_client: GAS*, ***, *commit_id: Optional[str] = None*)

Bases: object

This class defines the basic concept of the dataset client.

A *DatasetClientBase* contains the information needed for determining a unique dataset on TensorBay, and provides a series of method within dataset scope, such as *DatasetClientBase.list_segment_names()* and *DatasetClientBase.upload_catalog()*.

Parameters

- **name** – Dataset name.
- **dataset_id** – Dataset ID.
- **gas_client** – The initial client to interact between local and TensorBay.

checkout(*commit: Optional[str] = None*, *draft_number: Optional[int] = None*) → None

Checkout to commit or draft.

Parameters

- **commit** – The commit ID.
- **draft_number** – The draft number.

Raises **TypeError** – When both commit ID and draft number are provided or neither.

commit(*message: str*, ***, *tag: Optional[str] = None*) → None

Commit the draft.

Parameters

- **message** – The commit message.
- **tag** – A tag for current commit.

create_draft(*title: Optional[str] = None*) → int

Create the draft.

Parameters **title** – The draft title.

Returns The draft number of the created draft

property dataset_id

Return the TensorBay dataset ID.

Returns The TensorBay dataset ID.

delete_segment(*name: str*) → None

Delete a segment of the draft.

Parameters **name** – Segment name.

get_catalog() → *tensorbay.label.catalog.Catalog*

Get the catalog of the certain commit.

Returns Required *Catalog*.

list_draft_titles_and_numbers(***, *start: int = 0*, *stop: int = 9223372036854775807*) → Iterator[Dict[str, str]]

List the dict containing title and number of drafts.

Parameters

- **start** – The index to start.
- **stop** – The index to end.

Yields The dict containing title and number of drafts.

list_segment_names(*, *start: int = 0*, *stop: int = 9223372036854775807*) → Iterator[str]
List all segment names in a certain commit.

Parameters

- **start** – The index to start.
- **stop** – The index to end.

Yields Required segment names.

property name
Return the TensorBay dataset name.

Returns The TensorBay dataset name.

property status
Return the status of the dataset client.

Returns The status of the dataset client.

upload_catalog(*catalog:* tensorbay.label.catalog.Catalog) → None
Upload a catalog to the draft.

Parameters **catalog** – *Catalog* to upload.

Raises **TypeError** – When the catalog is empty.

tensorbay.client.exceptions

Classes refer to TensorBay exceptions.

Error	Description
GASResponseError	Post response error
GASDatasetError	The requested dataset does not exist
GASDatasetTypeError	The type of the requested dataset is wrong
GASDataTypeError	Dataset has multiple data types
GASLabelsetError	Requested data does not exist
GASLabelsetTypeError	The type of the requested data is wrong
GASSegmentError	The requested segment does not exist
GASPathError	Remote path does not follow linux style
GASFrameError	Uploading frame has no timestamp and no frame index.

exception tensorbay.client.exceptions.**GASDataTypeError**
Bases: *tensorbay.client.exceptions.GASException*

This error is raised to indicate that the dataset has multiple data types.

exception tensorbay.client.exceptions.**GASDatasetError**(*dataset_name: str*)
Bases: *tensorbay.client.exceptions.GASException*

This error is raised to indicate that the requested dataset does not exist.

Parameters **dataset_name** – The name of the missing dataset.

exception tensorbay.client.exceptions.**GASDatasetTypeError**(*dataset_name: str*, *is_fusion: bool*)

Bases: *tensorbay.client.exceptions.GASException*

This error is raised to indicate that the type of the requested dataset is wrong.

Parameters

- **dataset_name** – The name of the dataset whose requested type is wrong.
- **is_fusion** – Whether the dataset is a fusion dataset.

exception tensorbay.client.exceptions.**GASException**

Bases: Exception

This defines the parent class to the following specified error classes.

exception tensorbay.client.exceptions.**GASFrameError**

Bases: *tensorbay.client.exceptions.GASException*

This error is raised to indicate that uploading frame has no timestamp and no frame index.

exception tensorbay.client.exceptions.**GASLabelsetError**(*labelset_id: str*)

Bases: *tensorbay.client.exceptions.GASException*

This error is raised to indicate that requested data does not exist.

Parameters **labelset_id** – The labelset ID of the missing labelset.

exception tensorbay.client.exceptions.**GASLabelsetTypeError**(*labelset_id: str*, *is_fusion: bool*)

Bases: *tensorbay.client.exceptions.GASException*

This error is raised to indicate that the type of the requested labelset is wrong.

Parameters

- **labelset_id** – The ID of the labelset whose requested type is wrong.
- **is_fusion** – whether the labelset is a fusion labelset.

exception tensorbay.client.exceptions.**GASPathError**(*remote_path: str*)

Bases: *tensorbay.client.exceptions.GASException*

This error is raised to indicate that remote path does not follow linux style.

Parameters **remote_path** – The invalid remote path.

exception tensorbay.client.exceptions.**GASResponseError**(*response: requests.models.Response*)

Bases: *tensorbay.client.exceptions.GASException*

This error is raised to indicate post response error.

Parameters **response** – The response of the request.

exception tensorbay.client.exceptions.**GASSegmentError**(*segment_name: str*)

Bases: *tensorbay.client.exceptions.GASException*

This error is raised to indicate that the requested segment does not exist.

Parameters **segment_name** – The name of the missing segment_name.

tensorbay.client.gas

Class GAS.

The *GAS* defines the initial client to interact between local and TensorBay. It provides some operations on datasets level such as *GAS.create_dataset()*, *GAS.list_dataset_names()* and *GAS.get_dataset()*.

AccessKey is required when operating with dataset.

class tensorbay.client.gas.**GAS**(*access_key: str*, *url: str = ''*)

Bases: object

GAS defines the initial client to interact between local and TensorBay.

GAS provides some operations on dataset level such as *GAS.create_dataset()* *GAS.list_dataset_names()* and *GAS.get_dataset()*.

Parameters

- **access_key** – User's access key.
- **url** – The host URL of the gas website.

create_dataset(*name: str*, *is_fusion: typing_extensions.Literal[False] = False*, ***, *region: Optional[str] = 'None'*) → *tensorbay.client.dataset.DatasetClient*

create_dataset(*name: str*, *is_fusion: typing_extensions.Literal[True]*, ***, *region: Optional[str] = 'None'*) → tensorbay.client.dataset.FusionDatasetClient

create_dataset(*name: str*, *is_fusion: bool = False*, ***, *region: Optional[str] = 'None'*) → Union[*tensorbay.client.dataset.DatasetClient*, tensorbay.client.dataset.FusionDatasetClient]

Create a TensorBay dataset with given name.

Parameters

- **name** – Name of the dataset, unique for a user.
- **is_fusion** – Whether the dataset is a fusion dataset, True for fusion dataset.
- **region** – Region of the dataset to be stored, only support "beijing", "hangzhou", "shanghai", default is "shanghai".

Returns

The created *DatasetClient* instance or FusionDatasetClient instance (is_fusion=True), and the status of dataset client is "commit".

delete_dataset(*name: str*) → None

Delete a TensorBay dataset with given name.

Parameters **name** – Name of the dataset, unique for a user.

get_dataset(*name: str*, *is_fusion: typing_extensions.Literal[False] = False*) → *tensorbay.client.dataset.DatasetClient*

get_dataset(*name: str*, *is_fusion: typing_extensions.Literal[True]*) → tensorbay.client.dataset.FusionDatasetClient

get_dataset(*name: str*, *is_fusion: bool = False*) → Union[*tensorbay.client.dataset.DatasetClient*, tensorbay.client.dataset.FusionDatasetClient]

Get a TensorBay dataset with given name and commit ID.

Parameters

- **name** – The name of the requested dataset.
- **is_fusion** – Whether the dataset is a fusion dataset, True for fusion dataset.

Returns

The requested *DatasetClient* instance or `FusionDatasetClient` instance (is_fusion=True), and the status of dataset client is "commit".

Raises ***GASDatasetTypeError*** – When the requested dataset type is not the same as given.

list_dataset_names(*, *start: int = 0*, *stop: int = 9223372036854775807*) → Iterator[str]

List names of all TensorBay datasets.

Parameters

- **start** – The index to start.
- **stop** – The index to stop.

Yields Names of all datasets.

rename_dataset(*name: str*, *new_name: str*) → None

Rename a TensorBay Dataset with given name.

Parameters

- **name** – Name of the dataset, unique for a user.
- **new_name** – New name of the dataset, unique for a user.

upload_dataset(*dataset:* tensorbay.dataset.dataset.Dataset, *draft_number: Optional[int] = None*, *, *jobs: int = '1'*, *skip_uploaded_files: bool = 'False'*) → *tensorbay.client.dataset.DatasetClient*

upload_dataset(*dataset: tensorbay.dataset.dataset.FusionDataset*, *draft_number: Optional[int] = None*, *, *jobs: int = '1'*, *skip_uploaded_files: bool = 'False'*) → tensorbay.client.dataset.FusionDatasetClient

upload_dataset(*dataset: Union[*tensorbay.dataset.dataset.Dataset*, tensorbay.dataset.dataset.FusionDataset]*, *draft_number: Optional[int] = None*, *, *jobs: int = '1'*, *skip_uploaded_files: bool = 'False'*) → Union[*tensorbay.client.dataset.DatasetClient*, tensorbay.client.dataset.FusionDatasetClient]

Upload a local dataset to TensorBay.

This function will upload all information contains in the *Dataset* or `FusionDataset`, which includes:

- Create a TensorBay dataset with the name and type of input local dataset.
- **Upload all *Segment*** or `FusionSegment` in the dataset to TensorBay.

Parameters

- **dataset** – The *Dataset* or `FusionDataset` needs to be uploaded.
- **jobs** – The number of the max workers in multi-thread upload.
- **skip_uploaded_files** – Set it to True to skip the uploaded files.
- **draft_number** – The draft number.

Returns

The *DatasetClient* or `FusionDatasetClient` bound with the uploaded dataset.

tensorbay.client.log

Logging utility functions.

Dump_request_and_response dumps http request and response.

class tensorbay.client.log.**RequestLogging**(*request: requests.models.PreparedRequest*)
Bases: object

This class used to lazy load request to logging.

Parameters **request** – The request of the request.

class tensorbay.client.log.**ResponseLogging**(*response: requests.models.Response*)
Bases: object

This class used to lazy load response to logging.

Parameters **response** – The response of the request.

tensorbay.client.log.**dump_request_and_response**(*response: requests.models.Response*) → str

Dumps http request and response.

Parameters **response** – Http response and response.

Returns

Http request and response for logging, sample:

```
========================================================================
######################### HTTP Request ##########################
"url": https://gas.graviti.cn/gatewayv2/content-store/putObject
"method": POST
"headers": {
  "User-Agent": "python-requests/2.23.0",
  "Accept-Encoding": "gzip, deflate",
  "Accept": "*/*",
  "Connection": "keep-alive",
  "X-Token": "c3b1808b21024eb38f066809431e5bb9",
  "Content-Type": "multipart/form-data;␣
↪boundary=5adff1fc0524465593d6a9ad68aad7f9",
  "Content-Length": "330001"
}
"body":
--5adff1fc0524465593d6a9ad68aad7f9
b'Content-Disposition: form-data; name="contentSetId"\r\n\r\n'
b'e6110ff1-9e7c-4c98-aaf9-5e35522969b9'

--5adff1fc0524465593d6a9ad68aad7f9
b'Content-Disposition: form-data; name="filePath"\r\n\r\n'
b'4.jpg'

--5adff1fc0524465593d6a9ad68aad7f9
b'Content-Disposition: form-data; name="fileData"; filename="4.jpg"\r\
↪n\r\n'
[329633 bytes of object data]

--5adff1fc0524465593d6a9ad68aad7f9--

######################### HTTP Response ##########
"url": https://gas.graviti.cn/gatewayv2/content-stor
```

(continues on next page)

(continued from previous page)

```
"status_code": 200
"reason": OK
"headers": {
  "Date": "Sat, 23 May 2020 13:05:09 GMT",
  "Content-Type": "application/json;charset=utf-8",
  "Content-Length": "69",
  "Connection": "keep-alive",
  "Access-Control-Allow-Origin": "*",
  "X-Kong-Upstream-Latency": "180",
  "X-Kong-Proxy-Latency": "112",
  "Via": "kong/2.0.4"
}
"content": {
  "success": true,
  "code": "DATACENTER-0",
  "message": "success",
  "data": {}
}
==================================================
```

tensorbay.client.requests

Class Client and method multithread_upload.

Client can send POST, PUT, and GET requests to the TensorBay Dataset Open API.

multithread_upload() creates a multi-thread framework for uploading.

class tensorbay.client.requests.**Client**(*access_key: str*, *url: str = ''*)

Bases: object

This class defines *Client*.

Client defines the client that saves the user and URL information and supplies basic call methods that will be used by derived clients, such as sending GET, PUT and POST requests to TensorBay Open API.

Parameters

- **access_key** – User's access key.
- **url** – The URL of the graviti gas website.

do(*method: str*, *url: str*, ***kwargs: Any*) → requests.models.Response

Send a request.

Parameters

- **method** – The method of the request.
- **url** – The URL of the request.
- ****kwargs** – Extra keyword arguments to send in the GET request.

Returns Response of the request.

open_api_do(*method: str*, *section: str*, *dataset_id: str = ''*, ***kwargs: Any*) → requests.models.Response

Send a request to the TensorBay Open API.

Parameters

- **method** – The method of the request.

- **section** – The section of the request.
- **dataset_id** – Dataset ID.
- ****kwargs** – Extra keyword arguments to send in the POST request.

Returns Response of the request.

class tensorbay.client.requests.**Config**(*max_retries: int = 3*, *timeout: int = 15*, *is_intern: bool = False*)

Bases: object

This is a base class defining the concept of Post Config.

Parameters

- **max_retries** – Maximum retry times of the post request.
- **timeout** – Timeout value of the post request in seconds.
- **is_intern** – Whether the post request is from intern.

property is_intern

Get whether the post request is from intern.

Returns Whether the post request is from intern.

class tensorbay.client.requests.**TimeoutHTTPAdapter**(**args: Any*, *timeout: Optional[int] = None*, ***kwargs: Any*)

Bases: requests.adapters.HTTPAdapter

This class defines the http adapter for setting the timeout value.

Parameters

- ***args** – Extra arguments to initialize TimeoutHTTPAdapter.
- **timeout** – Timeout value of the post request in seconds.
- ****kwargs** – Extra keyword arguments to initialize TimeoutHTTPAdapter.

send(*request: requests.models.PreparedRequest*, *stream: Any = False*, *timeout: Optional[Any] = None*, *verify: Any = True*, *cert: Optional[Any] = None*, *proxies: Optional[Any] = None*) → Any

Send the request.

Parameters

- **request** – The PreparedRequest being sent.
- **stream** – Whether to stream the request content.
- **timeout** – Timeout value of the post request in seconds.
- **verify** – A path string to a CA bundle to use or a boolean which controls whether to verify the server's TLS certificate.
- **cert** – User-provided SSL certificate.
- **proxies** – Proxies dict applying to the request.

Returns Response object.

class tensorbay.client.requests.**UserSession**

Bases: requests.sessions.Session

This class defines UserSession.

request(*method: str*, *url: str*, **args: Any*, ***kwargs: Any*) → requests.models.Response

Make the request.

Parameters

- **method** – Method for the request.
- **url** – URL for the request.
- ***args** – Extra arguments to make the request.
- ****kwargs** – Extra keyword arguments to make the request.

Returns Response of the request.

Raises *GASResponseError* – If post response error.

tensorbay.client.requests.**multithread_upload**(*function: Callable[[_T], None]*, *arguments: Iterable[_T]*, *, *jobs: int = 1*) → None

Multi-thread upload framework.

Parameters

- **function** – The upload function.
- **arguments** – The arguments of the upload function.
- **jobs** – The number of the max workers in multi-thread uploading procession.

tensorbay.client.requests.**paging_range**(*start: int*, *stop: int*, *limit: int*) → Iterator[Tuple[int, int]]

A Generator which generates offset and limit for paging request.

Examples

```
>>> paging_range(0, 10, 3)
<generator object paging_range at 0x11b9932e0>
```

```
>>> list(paging_range(0, 10, 3))
[(0, 3), (3, 3), (6, 3), (9, 1)]
```

Parameters

- **start** – The paging index to start.
- **stop** – The paging index to end.
- **limit** – The paging limit.

Yields The tuple (offset, limit) for paging request.

tensorbay.client.segment

SegmentClientBase, SegmentClient and FusionSegmentClient.

The *SegmentClient* is a remote concept. It contains the information needed for determining a unique segment in a dataset on TensorBay, and provides a series of methods within a segment scope, such as *SegmentClient.upload_label()*, *SegmentClient.upload_data()*, *SegmentClient.list_data()* and so on. In contrast to the *SegmentClient*, *Segment* is a local concept. It represents a segment created locally. Please refer to *Segment* for more information.

Similarly to the *SegmentClient*, the *FusionSegmentClient* represents the fusion segment in a fusion dataset on TensorBay, and its local counterpart is FusionSegment. Please refer to FusionSegment for more information.

class tensorbay.client.segment.**FusionSegmentClient**(*name: str*, *data_client: FusionDatasetClient*)

Bases: *tensorbay.client.segment.SegmentClientBase*

This class defines *FusionSegmentClient*.

FusionSegmentClient inherits from *SegmentClientBase* and provides methods within a fusion segment scope, such as *FusionSegmentClient.upload_sensor()*, *FusionSegmentClient.upload_frame()* and *FusionSegmentClient.list_frames()*.

In contrast to *SegmentClient*, *FusionSegmentClient* has multiple sensors.

delete_sensor(*sensor_name: str*) → None

Delete a TensorBay sensor of the draft with the given sensor name.

Parameters **sensor_name** – The TensorBay sensor to delete.

list_frames(***, *start: int = 0*, *stop: int = 9223372036854775807*) → Iterator[tensorbay.dataset.frame.Frame]

List required frames in the segment in a certain commit.

Parameters

- **start** – The index to start.
- **stop** – The index to stop.

Yields Required `Frame`.

list_sensors() → Iterator[Union[Radar, Lidar, FisheyeCamera, Camera]]

List required sensor object in a segment client.

Yields Required sensor objects.

upload_frame(*frame: tensorbay.dataset.frame.Frame*, *timestamp: Optional[float] = None*) → None

Upload frame to the draft.

Parameters

- **frame** – The `Frame` to upload.
- **timestamp** – The mark to sort frames, supporting timestamp and float.

Raises

- ***GASPathError*** – When remote_path does not follow linux style.
- ***GASException*** – When uploading frame failed.
- **TypeError** – When frame id conflicts `

upload_sensor(*sensor:* tensorbay.sensor.sensor.Sensor) → None

Upload sensor to the draft.

Parameters **sensor** – The sensor to upload.

class tensorbay.client.segment.**SegmentClient**(*name: str*, *data_client: DatasetClient*)

Bases: *tensorbay.client.segment.SegmentClientBase*

This class defines *SegmentClient*.

SegmentClient inherits from SegmentClientBase and provides methods within a segment scope, such as *upload_label()*, *upload_data()*, *list_data()* and so on. In contrast to FusionSegmentClient, *SegmentClient* has only one sensor.

list_data(***, *start: int = 0*, *stop: int = 9223372036854775807*) → Iterator[*tensorbay.dataset.data.RemoteData*]

List required Data object in a dataset segment.

Parameters

- **start** – The index to start.
- **stop** – The index to stop.

Yields Required Data object.

list_data_paths(*, *start: int = 0*, *stop: int = 9223372036854775807*) → Iterator[str]
List required data path in a segment in a certain commit.

Parameters

- **start** – The index to start.
- **stop** – The index to end.

Yields Required data paths.

upload_data(*data:* tensorbay.dataset.data.Data) → None
Upload Data object to the draft.

Parameters **data** – The *Data*.

upload_file(*local_path: str*, *target_remote_path: str = ''*) → None
Upload data with local path to the draft.

Parameters

- **local_path** – The local path of the data to upload.
- **target_remote_path** – The path to save the data in segment client.

Raises

- ***GASPathError*** – When target_remote_path does not follow linux style.
- ***GASException*** – When uploading data failed.

upload_label(*data:* tensorbay.dataset.data.Data) → None
Upload label with Data object to the draft.

Parameters **data** – The data object which represents the local file to upload.

class tensorbay.client.segment.**SegmentClientBase**(*name: str*, *dataset_client: Union[DatasetClient, FusionDatasetClient]*)
Bases: object

This class defines the basic concept of *SegmentClient*.

A *SegmentClientBase* contains the information needed for determining a unique segment in a dataset on TensorBay.

Parameters

- **name** – Segment name.
- **dataset_client** – The dataset client.

delete_data(*remote_paths: Union[str, Iterable[str]]*) → None
Delete data of a segment in a certain commit with the given remote paths.

Parameters **remote_paths** – The remote paths of data in a segment.

property name
Return the segment name.

Returns The segment name.

property status
Return the status of the dataset client.

Returns The status of the dataset client.

1.10.2 tensorbay.dataset

tensorbay.dataset.data

Data.

Data is the most basic data unit of a *Dataset*. It contains path information of a data sample and its corresponding labels.

class tensorbay.dataset.data.**Data**(*local_path: str*, ***, *target_remote_path: Optional[str] = None*, *timestamp: Optional[float] = None*)
Bases: *tensorbay.dataset.data.DataBase*

Data is a combination of a specific local file and its label.

It contains the file local path, label information of the file and the file metadata, such as timestamp.

A Data instance contains one or several types of labels.

Parameters

- **local_path** – The file local path.
- **target_remote_path** – The file remote path after uploading to tensorbay.
- **timestamp** – The timestamp for the file.

path
The file local path.

timestamp
The timestamp for the file.

labels
The Labels that contains all the label information of the file.

dumps() → Dict[str, Any]
Dumps the local data into a dict.

Returns

Dumped data dict, which looks like:

```
{
    "localPath": <str>,
    "timestamp": <float>,
    "label": {
        "CLASSIFICATION": {...},
        "BOX2D": {...},
        "BOX3D": {...},
        "POLYGON2D": {...},
        "POLYLINE2D": {...},
        "KEYPOINTS2D": {...},
        "SENTENCE": {...}
```

(continues on next page)

(continued from previous page)

```
    }
}
```

classmethod loads(*contents: Dict[str, Any]*) → _T

Loads *Data* from a dict containing local data information.

Parameters **contents** – A dict containing the information of the data, which looks like:

```
{
    "localPath": <str>,
    "timestamp": <float>,
    "label": {
        "CLASSIFICATION": {...},
        "BOX2D": {...},
        "BOX3D": {...},
        "POLYGON2D": {...},
        "POLYLINE2D": {...},
        "KEYPOINTS2D": {...},
        "SENTENCE": {...}
    }
}
```

Returns A *Data* instance containing information from the given dict.

open() → _io.BufferedReader

Return the binary file pointer of this file.

The local file pointer will be obtained by build-in `open()`.

Returns The local file pointer for this data.

property target_remote_path

Return the target remote path of the data.

Target remote path will be used when this data is uploaded to tensorbay, and the target remote path will be the uploaded file's remote path.

Returns The target remote path of the data.

class tensorbay.dataset.data.**DataBase**(*path: str*, *, *timestamp: Optional[float] = None*)

Bases: *tensorbay.utility.repr.ReprMixin*

DataBase is a base class for the file and label combination.

Parameters

- **path** – The file path.
- **timestamp** – The timestamp for the file.

path

The file path.

timestamp

The timestamp for the file.

labels

The `Labels` that contains all the label information of the file.

static loads(*contents: Dict[str, Any]*) → _Type

Loads *Data* or *RemoteData* from a dict containing data information.

Parameters **contents** – A dict containing the information of the data, which looks like:

```
{
    "localPath" or "remotePath": <str>,
    "timestamp": <float>,
    "label": {
        "CLASSIFICATION": {...},
        "BOX2D": {...},
        "BOX3D": {...},
        "POLYGON2D": {...},
        "POLYLINE2D": {...},
        "KEYPOINTS2D": {...},
        "SENTENCE": {...}
    }
}
```

Returns A *Data* or *RemoteData* instance containing the given dict information.

class tensorbay.dataset.data.**RemoteData**(*remote_path: str*, ***, *timestamp: Optional[float] = None*, *url_getter: Optional[Callable[[str], str]] = None*)

Bases: *tensorbay.dataset.data.DataBase*

RemoteData is a combination of a specific tensorbay dataset file and its label.

It contains the file remote path, label information of the file and the file metadata, such as timestamp.

A RemoteData instance contains one or several types of labels.

Parameters

- **remote_path** – The file remote path.
- **timestamp** – The timestamp for the file.
- **url_getter** – The url getter of the remote file.

path
The file remote path.

timestamp
The timestamp for the file.

labels
The Labels that contains all the label information of the file.

dumps() → Dict[str, Any]
Dumps the remote data into a dict.

Returns

Dumped data dict, which looks like:

```
{
    "remotePath": <str>,
    "timestamp": <float>,
    "label": {
        "CLASSIFICATION": {...},
        "BOX2D": {...},
        "BOX3D": {...},
        "POLYGON2D": {...},
        "POLYLINE2D": {...},
        "KEYPOINTS2D": {...},
        "SENTENCE": {...}
```

(continues on next page)

(continued from previous page)

```
    }
}
```

get_url() → str
Return the url of the data hosted by tensorbay.

Returns The url of the data.

Raises **ValueError** – When the url_getter is missing.

classmethod loads(*contents: Dict[str, Any]*) → _T
Loads *RemoteData* from a dict containing remote data information.

Parameters **contents** – A dict containing the information of the data, which looks like:

```
{
    "remotePath": <str>,
    "timestamp": <float>,
    "label": {
        "CLASSIFICATION": {...},
        "BOX2D": {...},
        "BOX3D": {...},
        "POLYGON2D": {...},
        "POLYLINE2D": {...},
        "KEYPOINTS2D": {...},
        "SENTENCE": {...}
    }
}
```

Returns A *Data* instance containing information from the given dict.

open() → http.client.HTTPResponse
Return the binary file pointer of this file.

The remote file pointer will be obtained by `urllib.request.urlopen()`.

Returns The remote file pointer for this data.

tensorbay.dataset.dataset

DatasetBase, Dataset and FusionDataset.

DatasetBase defines the basic concept of a dataset, which is the top-level structure to handle your data files, labels and other additional information.

It represents a whole dataset contains several segments and is the base class of *Dataset* and `FusionDataset`.

Dataset is made up of data collected from only one sensor or data without sensor information. It consists of a list of *Segment*.

`FusionDataset` is made up of data collected from multiple sensors. It consists of a list of `FusionSegment`.

class tensorbay.dataset.dataset.**Dataset**(*name: str*, *is_continuous: bool = False*)
Bases: *tensorbay.dataset.dataset.DatasetBase*[*tensorbay.dataset.segment.Segment*]

This class defines the concept of dataset.

Dataset is made up of data collected from only one sensor or data without sensor information. It consists of a list of *Segment*.

create_segment(*segment_name: str = ''*) → *tensorbay.dataset.segment.Segment*
Create a segment with the given name.

Parameters **segment_name** – The name of the segment to create, which default value is an empty string.

Returns The created *Segment*.

class tensorbay.dataset.dataset.**DatasetBase**(*name: str*, *is_continuous: bool = False*)
Bases: *tensorbay.utility.name.NameMixin*, Sequence[tensorbay.dataset.dataset._T]

This class defines the concept of a basic dataset.

DatasetBase represents a whole dataset contains several segments and is the base class of *Dataset* and FusionDataset.

A dataset with labels should contain a *Catalog* indicating all the possible values of the labels.

Parameters

- **name** – The name of the dataset.
- **is_continuous** – Whether the data inside the dataset is time-continuous.

add_segment(*segment: _T*) → None
Add a segment to the dataset.

Parameters **segment** – The segment to be added.

property catalog
Return the catalog of the dataset.

Returns The *Catalog* of the dataset.

get_segment_by_name(*name: str*) → _T
Return the segment corresponding to the given name.

Parameters **name** – The name of the request segment.

Returns The segment which matches the input name.

property is_continuous
Return whether the data in dataset is time-continuous or not.

Returns *True* if the data is time-continuous, otherwise *False*.

load_catalog(*filepath: str*) → None
Load catalog from a json file.

Parameters **filepath** – The path of the json file which contains the catalog information.

tensorbay.dataset.segment

Segment and FusionSegment.

Segment is a concept in *Dataset*. It is the structure that composes *Dataset*, and consists of a series of *Data* without sensor information.

Fusion segment is a concept in FusionDataset. It is the structure that composes FusionDataset, and consists of a list of Frame along with multiple *Sensor*.

class tensorbay.dataset.segment.**Segment**(*name: str = ''*, *client: Optional[DatasetClient] = None*)

Bases: *tensorbay.utility.name.NameMixin*, *tensorbay.utility.user.UserMutableSequence*[DataBase._Type]

This class defines the concept of segment.

Segment is a concept in *Dataset*. It is the structure that composes *Dataset*, and consists of a series of *Data* without sensor information.

If the segment is inside of a time-continuous *Dataset*, the time continuity of the data should be indicated by :meth`~graviti.dataset.data.Data.remote_path`.

Since *Segment* extends *UserMutableSequence*, its basic operations are the same as a list's.

To initialize a Segment and add a *Data* to it:

```
segment = Segment(segment_name)
segment.append(Data())
```

Parameters

- **name** – The name of the segment, whose default value is an empty string.
- **client** – The DatasetClient if you want to read the segment from tensorbay.

sort(**, key: Callable[[Union[Data, RemoteData]], Any] = <function Segment.<lambda>>, reverse: bool = False*) → None

Sort the list in ascending order and return None.

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the order of two equal elements is maintained).

Parameters

- **key** – If a key function is given, apply it once to each item of the segment, and sort them according to their function values in ascending or descending order. By default, the data within the segment is sorted by fileuri.
- **reverse** – The reverse flag can be set as True to sort in descending order.

1.10.3 tensorbay.geometry

tensorbay.geometry.box

Box2D, Box3D.

Box2D contains the information of a 2D bounding box, such as the coordinates, width and height. It provides *Box2D.iou()* to calculate the intersection over union of two 2D boxes.

Box3D contains the information of a 3D bounding box such as the transform, translation, rotation and size. It provides *Box3D.iou()* to calculate the intersection over union of two 3D boxes.

class tensorbay.geometry.box.**Box2D**(*xmin: float*, *ymin: float*, *xmax: float*, *ymax: float*)

Bases: *tensorbay.utility.user.UserSequence*[float]

This class defines the concept of Box2D.

Box2D contains the information of a 2D bounding box, such as the coordinates, width and height. It provides *Box2D.iou()* to calculate the intersection over union of two 2D boxes.

Parameters

- **xmin** – The x coordinate of the top-left vertex of the 2D box.
- **ymin** – The y coordinate of the top-left vertex of the 2D box.
- **xmax** – The x coordinate of the bottom-right vertex of the 2D box.
- **ymax** – The y coordinate of the bottom-right vertex of the 2D box.

area() → float
Return the area of the 2D box.

Returns The area of the 2D box.

property br
Return the bottom right point.

Returns The bottom right point.

dumps() → Dict[str, float]
Dumps a 2D box into a dict.

Returns A dict containing vertex coordinates of the box.

classmethod from_xywh(*x: float*, *y: float*, *width: float*, *height: float*) → _B2
Create a *Box2D* instance from the top-left vertex and the width and the height.

Parameters

- **x** – X coordinate of the top left vertex of the box.
- **y** – Y coordinate of the top left vertex of the box.
- **width** – Length of the box along the x axis.
- **height** – Length of the box along the y axis.

Returns The created *Box2D* instance.

property height
Return the height of the 2D box.

Returns The height of the 2D box.

static iou(*box1:* tensorbay.geometry.box.Box2D, *box2:* tensorbay.geometry.box.Box2D) → float
Calculate the intersection over union of two 2D boxes.

Parameters

- **box1** – A 2D box.
- **box2** – A 2D box.

Returns The intersection over union between the two input boxes.

classmethod loads(*contents: Dict[str, float]*) → _B2
Load a *Box2D* from a dict containing coordinates of the 2D box.

Parameters **contents** – A dict containing coordinates of a 2D box:

```
{
    "xmin": ...
    "ymin": ...
    "xmax": ...
    "ymax": ...
}
```

Returns The loaded *Box2D* object.

property tl
Return the top left point.

Returns The top left point.

property width
Return the width of the 2D box.

Returns The width of the 2D box.

property xmax
Return the maximum x coordinate.

Returns Maximum x coordinate.

property xmin
Return the minimum x coordinate.

Returns Minimum x coordinate.

property ymax
Return the maximum y coordinate.

Returns Maximum y coordinate.

property ymin
Return the minimum y coordinate.

Returns Minimum y coordinate.

class tensorbay.geometry.box.**Box3D**(*transform: Union[None, tensorbay.geometry.transform.Transform3D, Sequence[Sequence[float]], numpy.ndarray] = None*, **, translation: Iterable[float] = (0, 0, 0)*, *rotation: Union[Iterable[float], quaternion.quaternion] = (1, 0, 0, 0)*, *size: Iterable[float] = (0, 0, 0)*)
Bases: *tensorbay.utility.repr.ReprMixin*

This class defines the concept of Box3D.

Box3D contains the information of a 3D bounding box such as the transform, translation, rotation and size. It provides *Box3D.iou()* to calculate the intersection over union of two 3D boxes.

Parameters

- **transform** – A *Transform3D* object or a 4x4 or 3x4 transform matrix.
- **translation** – Translation in a sequence of [x, y, z].
- **rotation** – Rotation in a sequence of [w, x, y, z] or a 3x3 rotation matrix or a numpy quaternion object.
- **size** – Size in a sequence of [x, y, z].

dumps() → Dict[str, Dict[str, float]]
Dumps the 3D box into a dict.

Returns A dict containing translation, rotation and size information.

classmethod iou(*box1:* tensorbay.geometry.box.Box3D, *box2:* tensorbay.geometry.box.Box3D, *angle_threshold: float = 5*) → float
Calculate the intersection over union between two 3D boxes.

Parameters

- **box1** – A 3D box.

- **box2** – A 3D box.
- **angle_threshold** – The threshold of the relative angles between two input 3d boxes in degree.

Returns The intersection over union of the two 3D boxes.

classmethod loads(*contents: Dict[str, Dict[str, float]]*) → _B3

Load a *Box3D* from a dict containing the coordinates of the 3D box.

Parameters **contents** – A dict containing the coordinates of a 3D box:

```
{
    "size": {
        "x": ...
        "y": ...
        "z": ...
    },
    "translation": {
        "x": ...
        "y": ...
        "z": ...
    },
    "rotation": {
        "w": ...
        "x": ...
        "y": ...
        "z": ...
    }
}
```

Returns The loaded *Box3D* object.

property rotation

Return the rotation of the 3D box.

Returns The rotation of the 3D box.

property size

Return the size of the 3D box.

Returns The size of the 3D box.

property transform

Return the transform of the 3D box.

Returns The transform of the 3D box.

property translation

Return the translation of the 3D box.

Returns The translation of the 3D box.

volume() → float

Return the volume of the 3D box.

Returns The volume of the 3D box.

tensorbay.geometry.keypoint

Keypoints2D, Keypoint2D.

Keypoint2D contains the information of 2D keypoint, such as the coordinates and visible status(optional).

Keypoints2D contains a list of 2D keypoint and is based on *PointList2D*.

class tensorbay.geometry.keypoint.**Keypoint2D**(**args: float*, ***kwargs: float*)

Bases: *tensorbay.utility.user.UserSequence*[float]

This class defines the concept of Keypoint2D.

Keypoint2D contains the information of 2D keypoint, such as the coordinates and visible status(optional).

Parameters

- **x** – The x coordinate of the 2D keypoint.
- **y** – The y coordinate of the 2D keypoint.
- **v** – The visible status(optional) of the 2D keypoint.

```
keypoint2d = Keypoint2D(x=1.0, y=2.0)
keypoint2d = Keypoint2D(x=1.0, y=2.0, v=0)
keypoint2d = Keypoint2D(x=1.0, y=2.0, v=1)
keypoint2d = Keypoint2D(x=1.0, y=2.0, v=2)
```

Visible status can be "BINARY" or "TERNARY":

Visual Status	v = 0	v = 1	v = 2
BINARY	visible	invisible	
TERNARY	visible	occluded	invisible

dumps() → Dict[str, float]

Dumps the *Keypoint2D* into a dict.

Returns A dict containing coordinates and visible status(optional) of the 2D keypoint.

classmethod loads(*contents: Dict[str, float]*) → _T

Load a *Keypoint2D* from a dict containing coordinates of a 2D keypoint.

Parameters **contents** – A dict containing coordinates and visible status(optional) of a 2D keypoint:

```
{
    "x": ...
    "y": ...
    "v": ...
}
```

Returns The loaded *Keypoint2D* object.

property v

Return the visible status of the 2D keypoint.

Returns Visible status of the 2D keypoint.

class tensorbay.geometry.keypoint.**Keypoints2D**(*points: Optional[Iterable[Iterable[float]]] = None*)

Bases: *tensorbay.geometry.polygon.PointList2D*[*tensorbay.geometry.keypoint.Keypoint2D*]

This class defines the concept of Keypoints2D.

Keypoints2D contains a list of 2D keypoint and is based on *PointList2D*.

classmethod loads(*contents: List[Dict[str, float]]*) → _P
Load a *Keypoints2D* from a list of dict.

Parameters **contents** – A list of dictionaries containing 2D keypoint:

```
[
    {
        "x": ...
        "y": ...
        "v": ...            --- optional
    },
    ...
]
```

Returns The loaded *Keypoints2D* object.

tensorbay.geometry.polygon

PointList2D, Polygon2D.

PointList2D contains a list of 2D points.

Polygon contains the coordinates of the vertexes of the polygon and provides *Polygon2D.area()* to calculate the area of the polygon.

class tensorbay.geometry.polygon.**PointList2D**(*points: Optional[Iterable[Iterable[float]]] = None*)
Bases: *tensorbay.utility.user.UserMutableSequence*[tensorbay.geometry.polygon._T]

This class defines the concept of PointList2D.

PointList2D contains a list of 2D points.

Parameters **points** – A list of 2D points.

bounds() → *tensorbay.geometry.box.Box2D*
Calculate the bounds of point list.

Returns The bounds of point list.

dumps() → List[Dict[str, float]]
Dumps a *PointList2D* into a point list.

Returns A list of dictionaries containing the coordinates of the vertexes of the polygon within the point list.

classmethod loads(*contents: List[Dict[str, float]]*) → _P
Load a *PointList2D* from a list of dictionaries.

Parameters **contents** – A list of dictionaries containing the coordinates of the vertexes of the polygon:

```
[
    {
        "x": ...
        "y": ...
    },
```

(continues on next page)

(continued from previous page)

```
    ...
]
```

Returns The loaded *PointList2D* object.

class tensorbay.geometry.polygon.**Polygon2D**(*points: Optional[Iterable[Iterable[float]]] = None*)

Bases: *tensorbay.geometry.polygon.PointList2D*[*tensorbay.geometry.vector.Vector2D*]

This class defines the concept of Polygon2D.

`Polygon` contains the coordinates of the vertexes of the polygon and provides *Polygon2D.area()* to calculate the area of the polygon.

area() → float

Return the area of the polygon.

The area is positive if the rotating direction of the points is counterclockwise, and negative if clockwise.

Returns The area of the polygon.

classmethod loads(*contents: List[Dict[str, float]]*) → _P

Load a *Polygon2D* from a list of dictionaries.

Parameters **contents** – A list of dictionaries containing the coordinates of the vertexes of the polygon:

```
[
    {
        "x": ...
        "y": ...
    },
    ...
]
```

Returns The loaded *Polygon2D* object.

tensorbay.geometry.polyline

Polyline2D.

Polyline2D contains the coordinates of the vertexes of the polyline and provides a series of methods to operate on polyline, such as *Polyline2D.uniform_frechet_distance()* and *Polyline2D.similarity()*.

class tensorbay.geometry.polyline.**Polyline2D**(*points: Optional[Iterable[Iterable[float]]] = None*)

Bases: *tensorbay.geometry.polygon.PointList2D*[*tensorbay.geometry.vector.Vector2D*]

This class defines the concept of Polyline2D.

Polyline2D contains the coordinates of the vertexes of the polyline and provides a series of methods to operate on polyline, such as *Polyline2D.uniform_frechet_distance()* and *Polyline2D.similarity()*.

classmethod loads(*contents: List[Dict[str, float]]*) → _P

Load a *Polyline2D* from a list of dict.

Parameters **contents** – A list of dict containing the coordinates of the vertexes of the polyline:

```
[
    {
        "x": ...
        "y": ...
    },
    ...
]
```

Returns The loaded *Polyline2D* object.

static similarity(*polyline1: Sequence[Sequence[float]]*, *polyline2: Sequence[Sequence[float]]*) → float

Calculate the similarity between two polylines, range from 0 to 1.

Parameters

- **polyline1** – The first polyline consists of multiple points.
- **polyline2** – The second polyline consisting of multiple points.

Returns The similarity between the two polylines. The larger the value, the higher the similarity.

static uniform_frechet_distance(*polyline1: Sequence[Sequence[float]]*, *polyline2: Sequence[Sequence[float]]*) → float

Compute the maximum distance between two curves if walk on a constant speed on a curve.

Parameters

- **polyline1** – The first polyline consists of multiple points.
- **polyline2** – The second polyline consists of multiple points.

Returns The computed distance between the two polylines.

tensorbay.geometry.transform

Transform3D.

Transform3D contains the rotation and translation of a 3D transform. *Transform3D.translation* is stored as *Vector3D*, and *Transform3D.rotation* is stored as numpy quaternion.

class tensorbay.geometry.transform.**Transform3D**(*transform: Union[None, tensorbay.geometry.transform.Transform3D, Sequence[Sequence[float]], numpy.ndarray] = None*, ***, *translation: Iterable[float] = (0, 0, 0)*, *rotation: Union[Iterable[float], quaternion.quaternion] = (1, 0, 0, 0)*)

Bases: *tensorbay.utility.repr.ReprMixin*

This class defines the concept of Transform3D.

Transform3D contains rotation and translation of the 3D transform.

Parameters

- **transform** – A *Transform3D* or a 4x4 or 3x4 transform matrix.
- **translation** – Translation in a sequence of [x, y, z].
- **rotation** – Rotation in a sequence of [w, x, y, z] or numpy quaternion.

Raises **ValueError** – If the shape of the input matrix is not correct.

as_matrix() → numpy.ndarray
Return the transform as a 4x4 transform matrix.

Returns A 4x4 numpy array represents the transform matrix.

dumps() → Dict[str, Dict[str, float]]
Dumps the *Transform3D* into a dict.

Returns

A dict containing rotation and translation information of the *Transform3D*.

inverse() → _T
Return the inverse of the transform.

Returns A *Transform3D* object representing the inverse of this *Transform3D*.

classmethod loads(*contents: Dict[str, Dict[str, float]]*) → _T
Load a *Transform3D* from a dict containing rotation and translation.

Parameters **contents** – A dict containing rotation and translation of a 3D transform:

```
{
    "translation": {
        "x": ...
        "y": ...
        "z": ...
    },
    "rotation": {
        "w": ...
        "x": ...
        "y": ...
        "z": ...
    }
}
```

Returns The loaded *Transform3D* object.

property rotation
Return the rotation of the 3D transform.

Returns Rotation in numpy quaternion.

set_rotation(*rotation: Union[Iterable[float], quaternion.quaternion]*) → None
Set the rotation of the transform.

Parameters **rotation** – Rotation in a sequence of [w, x, y, z] or numpy quaternion.

set_translation(*x: float*, *y: float*, *z: float*) → None
Set the translation of the transform.

Parameters

- **x** – The x coordinate of the translation.
- **y** – The y coordinate of the translation.
- **z** – The z coordinate of the translation. .. code:: python

  transform.set_translation(x=1, y=2, z=3)

property translation
Return the translation of the 3D transform.

Returns Translation in *Vector3D*.

tensorbay.geometry.vector

Vector, Vector2D, Vector3D.

Vector is the base class of *Vector2D* and *Vector3D*. It contains the coordinates of a 2D vector or a 3D vector.

Vector2D contains the coordinates of a 2D vector, extending *Vector*.

Vector3D contains the coordinates of a 3D vector, extending *Vector*.

class tensorbay.geometry.vector.**Vector**(*x: float*, *y: float*, *z: Optional[float] = None*)

Bases: *tensorbay.utility.user.UserSequence*[float]

This class defines the basic concept of Vector.

Vector contains the coordinates of a 2D vector or a 3D vector.

Parameters

- **x** – The x coordinate of the vector.
- **y** – The y coordinate of the vector.
- **z** – The z coordinate of the vector.

```
vector2d = Vector(x=1, y=2)
vector3d = Vector(x=1, y=2, z=3)
```

static loads(*contents: Dict[str, float]*) → Union[*tensorbay.geometry.vector.Vector2D*, *tensorbay.geometry.vector.Vector3D*]

Loads a *Vector* from a dict containing coordinates of the vector.

Parameters **contents** – A dict containing coordinates of the vector:

```
{
    "x": ...
    "y": ...
}
or
{
    "x": ...
    "y": ...
    "z": ...
}
```

Returns The loaded *Vector2D* or *Vector3D* object.

class tensorbay.geometry.vector.**Vector2D**(**args: float*, ***kwargs: float*)

Bases: *tensorbay.utility.user.UserSequence*[float]

This class defines the concept of Vector2D.

Vector2D contains the coordinates of a 2D vector.

Parameters

- **x** – The x coordinate of the 2D vector.
- **y** – The y coordinate of the 2D vector.

dumps() → Dict[str, float]

Dumps the vector into a dict.

Returns A dict containing the vector coordinate.

classmethod loads(*contents: Dict[str, float]*) → _V2
Load a *Vector2D* object from a dict containing coordinates of a 2D vector.

Parameters **contents** – A dict containing coordinates of a 2D vector:

```
{
    "x": ...
    "y": ...
}
```

Returns The loaded *Vector2D* object.

property x
Return the x coordinate of the vector.

Returns X coordinate in float type.

property y
Return the y coordinate of the vector.

Returns Y coordinate in float type.

class tensorbay.geometry.vector.**Vector3D**(**args: float*, ***kwargs: float*)
Bases: *tensorbay.utility.user.UserSequence*[float]

This class defines the concept of Vector3D.

Vector3D contains the coordinates of a 3D Vector.

Parameters

- **x** – The x coordinate of the 3D vector.
- **y** – The y coordinate of the 3D vector.
- **z** – The z coordinate of the 3D vector.

dumps() → Dict[str, float]
Dumps the vector into a dict.

Returns A dict containing the vector coordinates.

classmethod loads(*contents: Dict[str, float]*) → _V3
Load a *Vector3D* object from a dict containing coordinates of a 3D vector.

Parameters **contents** – A dict contains coordinates of a 3D vector:

```
{
    "x": ...
    "y": ...
    "z": ...
}
```

Returns The loaded *Vector3D* object.

property x
Return the x coordinate of the vector.

Returns X coordinate in float type.

property y
Return the y coordinate of the vector.

Returns Y coordinate in float type.

property z
Return the z coordinate of the vector.

Returns Z coordinate in float type.

1.10.4 tensorbay.label

tensorbay.label.attributes

Items and AttributeInfo.

AttributeInfo represents the information of an attribute. It refers to the Json schema method to describe an attribute.

Items is the base class of *AttributeInfo*, representing the items of an attribute.

class tensorbay.label.attributes.**AttributeInfo**(*name: str*, ***, *type_: Union[str, None, Type[Optional[Union[list, bool, int, float, str]]], Iterable[Union[str, None, Type[Optional[Union[list, bool, int, float, str]]]]]] = ''*, *enum: Optional[Iterable[Optional[Union[str, float, bool]]]] = None*, *minimum: Optional[float] = None*, *maximum: Optional[float] = None*, *items: Optional[*tensorbay.label.attributes.Items*] = None*, *parent_categories: Union[None, str, Iterable[str]] = None*, *description: Optional[str] = None*)

Bases: *tensorbay.utility.name.NameMixin*, *tensorbay.label.attributes.Items*

This class represents the information of an attribute.

It refers to the Json schema method to describe an attribute.

Parameters

- **name** – The name of the attribute.
- **type** – The type of the attribute value, could be a single type or multi-types. The type must be within the followings:
  - array
  - boolean
  - integer
  - number
  - string
  - null
  - instance
- **enum** – All the possible values of an enumeration attribute.
- **minimum** – The minimum value of number type attribute.
- **maximum** – The maximum value of number type attribute.

- **items** – The items inside array type attributes.
- **parent_categories** – The parent categories of the attribute.
- **description** – The description of the attribute.

type
The type of the attribute value, could be a single type or multi-types.

enum
All the possible values of an enumeration attribute.

minimum
The minimum value of number type attribute.

maximum
The maximum value of number type attribute.

items
The items inside array type attributes.

parent_categories
The parent categories of the attribute.

description
The description of the attribute.

dumps() → Dict[str, Any]
Dumps the information of this attribute into a dict.

Returns A dict containing all the information of this attribute.

classmethod loads(*contents: Dict[str, Any]*) → _T
Load an AttributeInfo from a dict containing the attribute information.

Parameters **contents** – A dict containing the information of the attribute, whose format should be like:

```
{
    "name":
    "type":
    "enum": [...]
    "minimum":
    "maximum":
    "items": {
        "enum": [...],
        "type":
        "minimum":
        "maximum":
    },
    "description":
    "parentCategories": [...]
}
```

Returns The loaded *AttributeInfo* object.

class tensorbay.label.attributes.**Items**(*, *type_: Union[str, None, Type[Optional[Union[list, bool, int, float, str]]], Iterable[Union[str, None, Type[Optional[Union[list, bool, int, float, str]]]]]] = ''*, *enum: Optional[Iterable[Optional[Union[str, float, bool]]]] = None*, *minimum: Optional[float] = None*, *maximum: Optional[float] = None*, *items: Optional[*tensorbay.label.attributes.Items*] = None*)

Bases: *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

The base class of *AttributeInfo*, representing the items of an attribute.

When the value type of an attribute is array, the *AttributeInfo* would contain an 'items' field.

Parameters

- **type** – The type of the attribute value, could be a single type or multi-types. The type must be within the followings:
  - array
  - boolean
  - integer
  - number
  - string
  - null
  - instance
- **enum** – All the possible values of an enumeration attribute.
- **minimum** – The minimum value of number type attribute.
- **maximum** – The maximum value of number type attribute.
- **items** – The items inside array type attributes.

type
The type of the attribute value, could be a single type or multi-types.

enum
All the possible values of an enumeration attribute.

minimum
The minimum value of number type attribute.

maximum
The maximum value of number type attribute.

items
The items inside array type attributes.

Raises **TypeError** – When both enum and **type_** are absent or when **type_** is array and items is absent.

dumps() → Dict[str, Any]
Dumps the information of the items into a dict.

Returns A dict containing all the information of the items.

classmethod loads(*contents: Dict[str, Any]*) → _T
Load an Items from a dict containing the items information.

Parameters **contents** – A dict containing the information of the items, whose format should be like:

```
{
    "type":
    "enum": [...]
    "minimum":
    "maximum":
    "items": {
        "enum": [...],
        "type":
        "minimum":
        "maximum":
    }
}
```

Returns The loaded *Items* object.

tensorbay.label.basic

LabelType, SubcatalogBase, Label.

LabelType is an enumeration type which includes all the supported label types within *Label*.

Subcatalogbase is the base class for different types of subcatalogs, which defines the basic concept of Subcatalog.

A *Data* instance contains one or several types of labels, all of which are stored in label.

A subcatalog class extends *SubcatalogBase* and needed SubcatalogMixin classes.

Different label types correspond to different label classes classes.

Table 1.4: label classes

label classes	explaination
Classification	classification type of label
LabeledBox2D	2D bounding box type of label
LabeledBox3D	3D bounding box type of label
LabeledPolygon2D	2D polygon type of label
LabeledPolyline2D	2D polyline type of label
LabeledKeypoints2D	2D keypoints type of label
LabeledSentence	transcripted sentence type of label

class tensorbay.label.basic.**Label**
Bases: *tensorbay.utility.repr.ReprMixin*

This class defines label.

It contains growing types of labels referring to different tasks.

dumps() → Dict[str, Any]
Dumps all labels into a dict.

Returns

Dumped labels dict, which looks like:

```
{
    "CLASSIFICATION": {...},
    "BOX2D": {...},
    "BOX3D": {...},
    "POLYGON2D": {...},
    "POLYLINE2D": {...},
    "KEYPOINTS2D": {...},
    "SENTENCE": {...},
}
```

classmethod loads(*contents: Dict[str, Any]*) → _T
Loads data from a dict containing the labels information.

Parameters **contents** – A dict containing the labels information, which looks like:

```
{
    "CLASSIFICATION": {...},
    "BOX2D": {...},
    "BOX3D": {...},
    "POLYGON2D": {...},
    "POLYLINE2D": {...},
    "KEYPOINTS2D": {...},
    "SENTENCE": {...},
}
```

Returns A *Label* instance containing labels information from the given dict.

class tensorbay.label.basic.**LabelType**(*value*)
Bases: *tensorbay.utility.type.TypeEnum*

This class defines all the supported types within *Label*.

property subcatalog_type
Return the corresponding subcatalog class.

Each label type has a corresponding Subcatalog class.

Returns The corresponding subcatalog type.

class tensorbay.label.basic.**SubcatalogBase**(**args*, ***kwds*)
Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This is the base class for different types of subcatalogs.

It defines the basic concept of Subcatalog, which is the collection of the labels information. Subcatalog contains the features, fields and specific definitions of the labels.

The Subcatalog format varies by label type.

description
The description of the entire subcatalog.

dumps() → Dict[str, Any]
Dumps all the information of the subcatalog into a dict.

Returns A dict containing all the information of the subcatalog.

classmethod loads(*contents: Dict[str, Any]*) → _T
Loads a subcatalog from a dict containing the information of the subcatalog.

Parameters **contents** – A dict containing the information of the subcatalog.

Returns The loaded *SubcatalogBase* object.

tensorbay.label.catalog

Catalog.

Catalog is used to describe the types of labels contained in a *DatasetBase* and all the optional values of the label contents.

A *Catalog* contains one or several *SubcatalogBase*, corresponding to different types of labels.

Table 1.5: subcatalog classes

subcatalog classes	explaination
ClassificationSubcatalog	subcatalog for classification type of label
Box2DSubcatalog	subcatalog for 2D bounding box type of label
Box3DSubcatalog	subcatalog for 3D bounding box type of label
Keypoints2DSubcatalog	subcatalog for 2D polygon type of label
Polygon2DSubcatalog	subcatalog for 2D polyline type of label
Polyline2DSubcatalog	subcatalog for 2D keypoints type of label
SentenceSubcatalog	subcatalog for transcripted sentence type of label

class tensorbay.label.catalog.**Catalog**

Bases: *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the concept of catalog.

Catalog is used to describe the types of labels contained in a *DatasetBase* and all the optional values of the label contents.

A *Catalog* contains one or several *SubcatalogBase*, corresponding to different types of labels. Each of the *SubcatalogBase* contains the features, fields and the specific definitions of the labels.

dumps() → Dict[str, Any]

Dumps the catalog into a dict containing the information of all the subcatalog.

Returns A dict containing all the subcatalog information with their label types as keys.

classmethod loads(*contents: Dict[str, Any]*) → _T

Load a Catalog from a dict containing the catalog information.

Parameters **contents** – A dict containing all the information of the catalog.

Returns The loaded *Catalog* object.

tensorbay.label.label_box

LabeledBox2D ,LabeledBox3D, Box2DSubcatalog, Box3DSubcatalog.

Box2DSubcatalog defines the subcatalog for 2D box type of labels.

LabeledBox2D is the 2D bounding box type of label, which is often used for CV tasks such as object detection.

Box3DSubcatalog defines the subcatalog for 3D box type of labels.

LabeledBox3D is the 3D bounding box type of label, which is often used for object detection in 3D point cloud.

class tensorbay.label.label_box.**Box2DSubcatalog**(*is_tracking: bool = False*)

Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the subcatalog for 2D box type of labels.

Parameters **is_tracking** – A boolean value indicates whether the corresponding subcatalog contains tracking information.

description
The description of the entire 2D box subcatalog.

categories
All the possible categories in the corresponding dataset stored in a *NameOrderedDict* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.supports.CategoryInfo*]

category_delimiter
The delimiter in category values indicating parent-child relationship.

Type str

attributes
All the possible attributes in the corresponding dataset stored in a *NameOrderedDict* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking
Whether the Subcatalog contains tracking information.

class tensorbay.label.label_box.**Box3DSubcatalog**(*is_tracking: bool = False*)
Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the subcatalog for 3D box type of labels.

Parameters **is_tracking** – A boolean value indicates whether the corresponding subcatalog contains tracking information.

description
The description of the entire 3D box subcatalog.

categories
All the possible categories in the corresponding dataset stored in a *NameOrderedDict* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.supports.CategoryInfo*]

category_delimiter
The delimiter in category values indicating parent-child relationship.

Type str

attributes
All the possible attributes in the corresponding dataset stored in a *NameOrderedDict* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking
Whether the Subcatalog contains tracking information.

class tensorbay.label.label_box.**LabeledBox2D**(*xmin: float*, *ymin: float*, *xmax: float*, *ymax: float*, ***, *category: Optional[str] = None*, *attributes: Optional[Dict[str, Any]] = None*, *instance: Optional[str] = None*)
Bases: *tensorbay.utility.user.UserSequence*[float]

This class defines the concept of 2D bounding box label.

LabeledBox2D is the 2D bounding box type of label, which is often used for CV tasks such as object detection.

Parameters

- **xmin** – The x coordinate of the top-left vertex of the labeled 2D box.
- **ymin** – The y coordinate of the top-left vertex of the labeled 2D box.
- **xmax** – The x coordinate of the bottom-right vertex of the labeled 2D box.
- **ymax** – The y coordinate of the bottom-right vertex of the labeled 2D box.
- **category** – The category of the label.
- **attributes** – The attributs of the label.
- **instance** – The instance id of the label.

category
The category of the label.

Type str

attributes
The attributes of the label.

Type Dict[str, Any]

instance
The instance id of the label.

Type str

dumps() → Dict[str, Any]
Dumps the current 2D bounding box label into a dict.

Returns

A dict containing all the information of the 2D box label, whose format is like:

```
{
    "box2d": {
        "xmin": <float>
        "ymin": <float>
        "xmax": <float>
        "ymax": <float>
    },
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

classmethod from_xywh(*x: float*, *y: float*, *width: float*, *height: float*, ***, *category: Optional[str] = None*, *attributes: Optional[Dict[str, Any]] = None*, *instance: Optional[str] = None*) → _T
Create a *LabeledBox2D* instance from the top-left vertex, the width and height.

Parameters

- **x** – X coordinate of the top left vertex of the box.
- **y** – Y coordinate of the top left vertex of the box.
- **width** – Length of the box along the x axis.
- **height** – Length of the box along the y axis.
- **category** – The category of the label.
- **attributes** – The attributs of the label.
- **instance** – The instance id of the label.

Returns The created *LabeledBox2D* instance.

classmethod loads(*contents: Dict[str, Any]*) → _T

Loads a LabeledBox2D from a dict containing the information of the label.

Parameters **contents** – A dict containing the information of the 2D bounding box label, whose format should be like:

```
{
    "box2d": {
        "xmin": <float>
        "ymin": <float>
        "xmax": <float>
        "ymax": <float>
    },
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

Returns The loaded *LabeledBox2D* object.

class tensorbay.label.label_box.**LabeledBox3D**(*transform: Union[None, tensorbay.geometry.transform.Transform3D, Sequence[Sequence[float]], numpy.ndarray] = None*, ***, *translation: Iterable[float] = (0, 0, 0)*, *rotation: Union[Iterable[float], quaternion.quaternion] = (1, 0, 0, 0)*, *size: Iterable[float] = (0, 0, 0)*, *category: Optional[str] = None*, *attributes: Optional[Dict[str, Any]] = None*, *instance: Optional[str] = None*)

Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the concept of 3D bounding box label.

LabeledBox3D is the 3D bounding box type of label, which is often used for object detection in 3D point cloud.

Parameters

- **transform** – The transform of the 3D bounding box label in a *Transform3D* object or a 4x4 or 3x4 transformation matrix.

- **translation** – Translation of the 3D bounding box label in a sequence of [x, y, z].
- **rotation** – Rotation of the 3D bounding box label in a sequence of [w, x, y, z] or a 3x3 rotation matrix or a numpy quaternion object.
- **size** – Size of the 3D bounding box label in a sequence of [x, y, z].
- **category** – Category of the 3D bounding box label.
- **attributes** – Attributs of the 3D bounding box label.
- **instance** – The instance id of the 3D bounding box label.

category
The category of the label.

Type str

attributes
The attributes of the label.

Type Dict[str, Any]

instance
The instance id of the label.

Type str

size
The size of the 3D bounding box.

transform
The transform of the 3D bounding box.

dumps() → Dict[str, Any]
Dumps the current 3D bounding box label into a dict.

Returns

A dict containing all the information of the 3D bounding box label, whose format is like:

```
{
    "box3d": {
        "translation": {
            "x": <float>
            "y": <float>
            "z": <float>
        },
        "rotation": {
            "w": <float>
            "x": <float>
            "y": <float>
            "z": <float>
        },
        "size": {
            "x": <float>
            "y": <float>
            "z": <float>
        }
    },
    "category": <str>
    "attributes": {
        <key>: <value>
```

(continues on next page)

(continued from previous page)

```
        ...
        ...
    },
    "instance": <str>
},
```

classmethod loads(*contents: Dict[str, Any]*) → _T

Loads a LabeledBox3D from a dict containing the information of the label.

Parameters **contents** – A dict containing the information of the 3D bounding box label, whose format should be like:

```
{
    "box3d": {
        "translation": {
            "x": <float>
            "y": <float>
            "z": <float>
        },
        "rotation": {
            "w": <float>
            "x": <float>
            "y": <float>
            "z": <float>
        },
        "size": {
            "x": <float>
            "y": <float>
            "z": <float>
        }
    },
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

Returns The loaded *LabeledBox3D* object.

tensorbay.label.label_classification

Classification.

ClassificationSubcatalog defines the subcatalog for classification type of labels.

Classification defines the concept of classification label, which can apply to different types of data, such as images and texts.

class tensorbay.label.label_classification.**Classification**(*category: Optional[str] = None*, *attributes: Optional[Dict[str, Any]] = None*)

Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the concept of classification label.

Classification is the classification type of label, which applies to different types of data, such as images and texts.

Parameters

- **category** – The category of the label.
- **attributes** – The attributes of the label.

category
The category of the label.

Type str

attributes
The attributes of the label.

Type Dict[str, Any]

classmethod loads(*contents: Dict[str, Any]*) → _T
Loads a Classification label from a dict containing the label information.

Parameters **contents** – A dict containing the information of the classification label, whose format should be like:

```
{
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    }
}
```

Returns The loaded *Classification* object.

class tensorbay.label.label_classification.**ClassificationSubcatalog**(**args*, ***kwds*)

Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the subcatalog for classification type of labels.

description
The description of the entire classification subcatalog.

categories
All the possible categories in the corresponding dataset stored in a *NameOrderedDict* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.supports.CategoryInfo*]

category_delimiter
The delimiter in category values indicating parent-child relationship.

Type str

attributes
All the possible attributes in the corresponding dataset stored in a *NameOrderedDict* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.attributes.AttributeInfo*]

tensorbay.label.label_keypoints

LabeledKeypoints2D, Keypoints2DSubcatalog.

Keypoints2DSubcatalog defines the subcatalog for 2D keypoints type of labels.

LabeledKeypoints2D is the 2D keypoints type of label, which is often used for CV tasks such as human body pose estimation.

class tensorbay.label.label_keypoints.**Keypoints2DSubcatalog**(*is_tracking: bool = False*)

Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the subcatalog for 2D keypoints type of labels.

Parameters **is_tracking** – A boolean value indicates whether the corresponding subcatalog contains tracking information.

description
The description of the entire 2D keypoints subcatalog.

categories
All the possible categories in the corresponding dataset stored in a *NameOrderedDict* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.supports.CategoryInfo*]

category_delimiter
The delimiter in category values indicating parent-child relationship.

Type str

attributes
All the possible attributes in the corresponding dataset stored in a *NameOrderedDict* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking
Whether the Subcatalog contains tracking information.

add_keypoints(*number: int*, ***, *names: Optional[Iterable[str]] = None*, *skeleton: Optional[Iterable[Iterable[int]]] = None*, *visible: Optional[str] = None*, *parent_categories: Union[None, str, Iterable[str]] = None*, *description: Optional[str] = None*) → None
Add a type of keypoints to the subcatalog.

Parameters

- **number** – The number of keypoints.
- **names** – All the names of keypoints.
- **skeleton** – The skeleton of the keypoints indicating which keypoint should connect with another.
- **visible** – The visible type of the keypoints, can only be 'BINARY' or 'TERNARY'. It determines the range of the *Keypoint2D.v*.
- **parent_categories** – The parent categories of the keypoints.
- **description** – The description of keypoints.

dumps() → Dict[str, Any]
Dumps all the information of the keypoints into a dict.

Returns A dict containing all the information of this Keypoints2DSubcatalog.

property keypoints
Return the KeypointsInfo of the Subcatalog.

Returns A list of *KeypointsInfo*.

class tensorbay.label.label_keypoints.**LabeledKeypoints2D**(*keypoints: Optional[Iterable[Iterable[float]]] = None*, ***, *category: Optional[str] = None*, *attributes: Optional[Dict[str, Any]] = None*, *instance: Optional[str] = None*)
Bases: *tensorbay.geometry.polygon.PointList2D*[*tensorbay.geometry.keypoint.Keypoint2D*]

This class defines the concept of 2D keypoints label.

LabeledKeypoints2D is the 2D keypoints type of label, which is often used for CV tasks such as human body pose estimation.

Parameters

- **keypoints** – A list of 2D keypoint.
- **category** – The category of the label.
- **attributes** – The attributes of the label.
- **instance** – The instance id of the label.

category
The category of the label.

Type str

attributes
The attributes of the label.

Type Dict[str, Any]

instance
The instance id of the label.

Type str

dumps() → Dict[str, Any]
Dumps the current 2D keypoints label into a dict.

Returns

A dict containing all the information of the 2D keypoints label, whose format is like:

```
{
    "keypoints2d": [
        { "x": <float>
          "y": <float>
          "v": <int>
        },
```

(continues on next page)

(continued from previous page)

```
        ...
        ...
    ],
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

classmethod loads(*contents: Dict[str, Any]*) → _T

Loads a LabeledKeypoints2D from a dict containing the information of the label.

Parameters **contents** – A dict containing the information of the 2D keypoints label, whose format should be like:

```
{
    "keypoints2d": [
        { "x": <float>
          "y": <float>
          "v": <int>
        },
        ...
        ...
    ],
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

Returns The loaded *LabeledKeypoints2D* object.

tensorbay.label.label_polygon

LabeledPolygon2D, Polygon2DSubcatalog.

Polygon2DSubcatalog defines the subcatalog for 2D polygon type of labels.

LabeledPolygon2D is the 2D polygon type of label, which is often used for CV tasks such as semantic segmentation.

class tensorbay.label.label_polygon.**LabeledPolygon2D**(*points: Optional[Iterable[Iterable[float]]] = None*, ***, *category: Optional[str] = None*, *attributes: Optional[Dict[str, Any]] = None*, *instance: Optional[str] = None*)

Bases: *tensorbay.geometry.polygon.PointList2D*[*tensorbay.geometry.vector.Vector2D*]

This class defines the concept of polygon2D label.

LabeledPolygon2D is the 2D polygon type of label, which is often used for CV tasks such as semantic segmentation.

Parameters

- **points** – A list of 2D points representing the vertexes of the 2D polygon.
- **category** – The category of the label.
- **attributes** – The attributs of the label.
- **instance** – The instance id of the label.

category
The category of the label.

Type str

attributes
The attributes of the label.

Type Dict[str, Any]

instance
The instance id of the label.

Type str

dumps() → Dict[str, Any]
Dumps the current 2D polygon label into a dict.

Returns

A dict containing all the information of the 2D polygon label, whose format is like:

```
{
    "polygon": [
        { "x": <int>
          "y": <int>
        },
        ...
        ...
    ],
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

classmethod loads(*contents: Dict[str, Any]*) → _T
Loads a LabeledPolygon2D from a dict containing the information of the label.

Parameters **contents** – A dict containing the information of the 2D polygon label, whose format should be like:

```
{
    "polygon": [
        { "x": <int>
```

(continues on next page)

(continued from previous page)

```
            "y": <int>
        },
        ...
        ...
    ],
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

Returns The loaded *LabeledPolygon2D* object.

class tensorbay.label.label_polygon.**Polygon2DSubcatalog**(*is_tracking: bool = False*)

Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the subcatalog for 2D polygon type of labels.

Parameters **is_tracking** – A boolean value indicates whether the corresponding subcatalog contains tracking information.

description
The description of the entire 2D polygon subcatalog.

categories
All the possible categories in the corresponding dataset stored in a *NameOrderedDict* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.supports.CategoryInfo*]

category_delimiter
The delimiter in category values indicating parent-child relationship.

Type str

attributes
All the possible attributes in the corresponding dataset stored in a *NameOrderedDict* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking
Whether the Subcatalog contains tracking information.

tensorbay.label.label_polyline

LabeledPolyline2D, Polyline2DSubcatalog.

Polyline2DSubcatalog defines the subcatalog for 2D polyline type of labels.

LabeledPolyline2D is the 2D polyline type of label, which is often used for CV tasks such as lane detection.

class tensorbay.label.label_polyline.**LabeledPolyline2D**(*points: Optional[Iterable[Iterable[float]]] = None*, ***, *category: Optional[str] = None*, *attributes: Optional[Dict[str, Any]] = None*, *instance: Optional[str] = None*)

Bases: *tensorbay.geometry.polygon.PointList2D*[*tensorbay.geometry.vector.Vector2D*]

This class defines the concept of polyline2D label.

LabeledPolyline2D is the 2D polyline type of label, which is often used for CV tasks such as lane detection.

Parameters

- **points** – A list of 2D points representing the vertexes of the 2D polyline.
- **category** – The category of the label.
- **attributes** – The attributes of the label.
- **instance** – The instance id of the label.

category
The category of the label.

Type str

attributes
The attributes of the label.

Type Dict[str, Any]

instance
The instance id of the label.

Type str

dumps() → Dict[str, Any]
Dumps the current 2D polyline label into a dict.

Returns

A dict containing all the information of the 2D polyline label, whose format is like:

```
{
    "polyline": [
        { "x": <int>
          "y": <int>
        },
        ...
        ...
    ],
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

classmethod loads(*contents: Dict[str, Any]*) → _T

Loads a LabeledPolyline2D from a dict containing the information of the label.

Parameters **contents** – A dict containing the information of the 2D polyline label, whose format should be like:

```
{
    "polyline": [
        { "x": <int>
          "y": <int>
        },
        ...
        ...
    ],
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

Returns The loaded *LabeledPolyline2D* object.

class tensorbay.label.label_polyline.**Polyline2DSubcatalog**(*is_tracking: bool = False*)

Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the subcatalog for 2D polyline type of labels.

Parameters **is_tracking** – A boolean value indicates whether the corresponding subcatalog contains tracking information.

description

The description of the entire 2D polyline subcatalog.

categories

All the possible categories in the corresponding dataset stored in a *NameOrderedDict* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.supports.CategoryInfo*]

category_delimiter

The delimiter in category values indicating parent-child relationship.

Type str

attributes

All the possible attributes in the corresponding dataset stored in a *NameOrderedDict* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking

Whether the Subcatalog contains tracking information.

tensorbay.label.label_sentence

Word, LabeledSentence, SentenceSubcatalog.

SentenceSubcatalog defines the subcatalog for audio transcripted sentence type of labels.

Word is a word within a phonetic transcription sentence, containing the content of the word, the start and end time in the audio.

LabeledSentence is the transcripted sentence type of label. which is often used for tasks such as automatic speech recognition.

class tensorbay.label.label_sentence.**LabeledSentence**(*sentence: Optional[Iterable[tensorbay.label.label_sentence.Word]] = None*, *spell: Optional[Iterable[tensorbay.label.label_sentence.Word]] = None*, *phone: Optional[Iterable[tensorbay.label.label_sentence.Word]] = None*, ***, *attributes: Optional[Dict[str, Any]] = None*)

Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the concept of phonetic transcription lable.

LabeledSentence is the transcripted sentence type of label. which is often used for tasks such as automatic speech recognition.

Parameters

- **sentence** – A list of sentence.
- **spell** – A list of spell, only exists in Chinese language.
- **phone** – A list of phone.
- **attributes** – The attributes of the label.

sentence
The transcripted sentence.

spell
The spell within the sentence, only exists in Chinese language.

phone
The phone of the sentence label.

attributes
The attributes of the label.

Type Dict[str, Any]

dumps() → Dict[str, Any]
Dumps the current label into a dict.

Returns

A dict containing all the information of the sentence label, whose format is like:

```
{
    "sentence": [
        {
            "text":  <str>
```

(continues on next page)

(continued from previous page)

```
            "begin": <float>
            "end":   <float>
        }
        ...
        ...
    ],
    "spell": [
        {
            "text":  <str>
            "begin": <float>
            "end":   <float>
        }
        ...
        ...
    ],
    "phone": [
        {
            "text":  <str>
            "begin": <float>
            "end":   <float>
        }
        ...
        ...
    ],
    "attributes": {
        <key>: <value>,
        ...
        ...
    }
}
```

classmethod loads(*contents: Dict[str, Any]*) → _T

Loads a LabeledSentence from a dict containing the information of the label.

Parameters **contents** – A dict containing the information of the sentence label, whose format should be like:

```
{
    "sentence": [
        {
            "text":  <str>
            "begin": <float>
            "end":   <float>
        }
        ...
        ...
    ],
    "spell": [
        {
            "text":  <str>
            "begin": <float>
            "end":   <float>
        }
        ...
        ...
    ],
    "phone": [
```

(continues on next page)

(continued from previous page)

```
        {
            "text":  <str>
            "begin": <float>
            "end":   <float>
        }
        ...
        ...
    ],
    "attributes": {
        <key>: <value>,
        ...
        ...
    }
}
```

Returns The loaded *LabeledSentence* object.

class tensorbay.label.label_sentence.**SentenceSubcatalog**(*is_sample: bool = False*, *sample_rate: Optional[int] = None*, *lexicon: Optional[List[List[str]]] = None*)

Bases: *tensorbay.utility.type.TypeMixin*[*tensorbay.label.basic.LabelType*], *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the subcatalog for audio transcripted sentence type of labels.

Parameters

- **is_sample** – A boolen value indicates whether time format is sample related.
- **sample_rate** – The number of samples of audio carried per second.
- **lexicon** – A list consists all of text and phone.

description
The description of the entire sentence subcatalog.

is_sample
A boolen value indicates whether time format is sample related.

sample_rate
The number of samples of audio carried per second.

lexicon
A list consists all of text and phone.

attributes
All the possible attributes in the corresponding dataset stored in a *NameOrderedDict* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.attributes.AttributeInfo*]

Raises **TypeError** – When sample_rate is None and is_sample is True.

append_lexicon(*lexemes: List[str]*) → None
Add lexemes to lexicon.

Parameters **lexemes** – A list consists of text and phone.

dumps() → Dict[str, Any]
Dumps the information of this SentenceSubcatalog into a dict.

Returns A dict containing all information of this SentenceSubcatalog.

class tensorbay.label.label_sentence.**Word**(*text: str*, *begin: Optional[float] = None*, *end: Optional[float] = None*)
Bases: *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the concept of word.

Word is a word within a phonetic transcription sentence, containing the content of the word, the start and end time in the audio.

Parameters

- **text** – The content of the word.
- **begin** – The begin time of the word in the audio.
- **end** – The end time of the word in the audio.

text
The content of the word.

begin
The begin time of the word in the audio.

end
The end time of the word in the audio.

dumps() → Dict[str, Union[str, float]]
Dumps the current word into a dict.

Returns

A dict containing all the information of the word, whose format is like:

```
{
    "text": str ,
    "begin": float,
    "end": float,
}
```

classmethod loads(*contents: Dict[str, Union[str, float]]*) → _T
Loads a Word from a dict containing the information of the word.

Parameters **contents** – A dict containing the information of the word, whose format should be like:

```
{
    "text": str ,
    "begin": float,
    "end": float,
}
```

Returns The loaded *Word* object.

tensorbay.label.supports

CatagoryInfo, KeypointsInfo and different SubcatalogMixin classes.

`CatagoryInfo` defines a category with the name and description of it.

`KeypointsInfo` defines the structure of a set of keypoints.

`SubcatalogMixin` is the base class of different mixin classes for subcatalog.

Table 1.6: mixin classes for subcatalog

mixin classes for subcatalog	explaination
`IsTrackingMixin`	a mixin class supporting tracking information of a subcatalog
`CategoriesMixin`	a mixin class supporting category information of a subcatalog
`AttributesMixin`	a mixin class supporting attribute information of a subcatalog

class tensorbay.label.supports.**AttributesMixin**

Bases: *tensorbay.label.supports.SubcatalogMixin*

A mixin class supporting attribute information of a subcatalog.

attributes

All the possible attributes in the corresponding dataset stored in a *`NameOrderedDict`* with the attribute names as keys and the `AttributeInfo` as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.attributes.AttributeInfo*]

add_attribute(*name: str*, ***, *type_: Union[str, None, Type[Optional[Union[list, bool, int, float, str]]], Iterable[Union[str, None, Type[Optional[Union[list, bool, int, float, str]]]]]] = ''*, *enum: Optional[Iterable[Optional[Union[str, float, bool]]]] = None*, *minimum: Optional[float] = None*, *maximum: Optional[float] = None*, *items: Optional[*tensorbay.label.attributes.Items*] = None*, *parent_categories: Union[None, str, Iterable[str]] = None*, *description: Optional[str] = None*) → None

Add an attribute to the Subcatalog.

Parameters

- **name** – The name of the attribute.
- **type** – The type of the attribute value, could be a single type or multi-types. The type must be within the followings: - array - boolean - integer - number - string - null - instance
- **enum** – All the possible values of an enumeration attribute.
- **minimum** – The minimum value of number type attribute.
- **maximum** – The maximum value of number type attribute.
- **items** – The items inside array type attributes.
- **parent_categories** – The parent categories of the attribute.
- **description** – The description of the attributes.

class tensorbay.label.supports.**CategoriesMixin**

Bases: *tensorbay.label.supports.SubcatalogMixin*

A mixin class supporting category information of a subcatalog.

categories

All the possible categories in the corresponding dataset stored in a *`NameOrderedDict`* with the category names as keys and the *`CategoryInfo`* as values.

Type *tensorbay.utility.name.NameOrderedDict*[*tensorbay.label.supports.CategoryInfo*]

category_delimiter
The delimiter in category values indicating parent-child relationship.

Type str

add_category(*name: str*, *description: Optional[str] = None*) → None
Add a category to the Subcatalog.

Parameters

- **name** – The name of the category.
- **description** – The description of the category.

class tensorbay.label.supports.**CategoryInfo**(*name: str*, *description: Optional[str] = None*)
Bases: *tensorbay.utility.name.NameMixin*

This class represents the information of a category, including category name and description.

Parameters

- **name** – The name of the category.
- **description** – The description of the category.

name
The name of the category.

description
The description of the category.

dumps() → Dict[str, str]
Dumps the CatagoryInfo into a dict.

Returns

A dict containing the information in the CategoryInfo, whose format is like:

```
{
    "name": <str>
    "description": <str>
}
```

classmethod loads(*contents: Dict[str, str]*) → _T
Loads a CategoryInfo from a dict containing the category.

Parameters **contents** – A dict containing the information of the category, whose format should be like:

```
{
    "name": <str>
    "description": <str>
}
```

Returns The loaded *CategoryInfo* object.

class tensorbay.label.supports.**IsTrackingMixin**(*is_tracking: bool = False*)
Bases: *tensorbay.label.supports.SubcatalogMixin*

A mixin class supporting tracking information of a subcatalog.

Parameters **is_tracking** – Whether the Subcatalog contains tracking information.

is_tracking
Whether the Subcatalog contains tracking information.

class tensorbay.label.supports.**KeypointsInfo**(*number: int*, ***, *names: Optional[Iterable[str]] = None*, *skeleton: Optional[Iterable[Iterable[int]]] = None*, *visible: Optional[str] = None*, *parent_categories: Union[None, str, Iterable[str]] = None*, *description: Optional[str] = None*)

Bases: *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

This class defines the structure of a set of keypoints.

Parameters

- **number** – The number of the set of keypoints.
- **names** – All the names of the keypoints.
- **skeleton** – The skeleton of the keypoints indicating which keypoint should connect with another.
- **visible** – The visible type of the keypoints, can only be 'BINARY' or 'TERNARY'. It determines the range of the *Keypoint2D.v*.
- **parent_categories** – The parent categories of the keypoints.
- **description** – The description of the keypoints.

names
All the names of the keypoints.

skeleton
The skeleton of the keypoints indicating which keypoint should connect with another.

visible
The visible type of the keypoints, can only be 'BINARY' or 'TERNARY'. It determines the range of the *Keypoint2D.v*.

parent_categories
The parent categories of the keypoints.

description
The description of the keypoints.

dumps() → Dict[str, Any]
Dumps all the keypoint information into a dict.

Returns

A dict containing all the information of the keypoint, whose format is like:

```
{
    "number":
    "names": [...],
    "skeleton": [
        [<index>, <index>],
        ...
    ],
    "visible": "TERNARY" or "BINARY"
    "parentCategories": [...],
```

(continues on next page)

(continued from previous page)

```
    "description":
}
```

classmethod loads(*contents: Dict[str, Any]*) → _T

Loads a KeypointsInfo from a dict containing the information of the keypoints.

Parameters **contents** – A dict containing all the information of the set of keypoints, whose format should be like:

```
{
    "number":
    "names": [...],
    "skeleton": [
        [<index>, <index>],
        ...
    ],
    "visible": "TERNARY" or "BINARY"
    "parentCategories": [...],
    "description":
}
```

Returns The loaded *KeypointsInfo* object.

property number

Return the number of the keypoints.

Returns The number of the keypoints.

class tensorbay.label.supports.**SubcatalogMixin**

Bases: tensorbay.utility.common.EqMixin

The base class of different mixin classes for subcatalog.

1.10.5 tensorbay.opendataset

tensorbay.opendataset.AnimalPose.loader

Dataloader of 5 Categories AnimalPose dataset and 7 Categories AnimalPose dataset.

tensorbay.opendataset.AnimalPose.loader.**AnimalPose5**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of 5 Categories AnimalPose dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    keypoint_image_part1/
        cat/
            2007_000549.jpg
            2007_000876.jpg
            ...
        ...
    PASCAL2011_animal_annotation/
        cat/
            2007_000549_1.xml
            2007_000876_1.xml
            2007_000876_2.xml
```

(continues on next page)

(continued from previous page)

```
            ...
        ...
    animalpose_image_part2/
        cat/
            ca1.jpeg
            ca2.jpeg
            ...
        ...
    animalpose_anno2/
        cat/
            ca1.xml
            ca2.xml
        ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.AnimalPose.loader.**AnimalPose7**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of 7 Categories AnimalPose dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    bndbox_image/
        antelope/
            Img-77.jpg
            ...
        ...
    bndbox_anno/
        antelope.json
        ...
```

Returns loaded *Dataset* object.

tensorbay.opendataset.AnimalsWithAttributes2.loader

Dataloader of the Animals with attributes 2 dataset.

tensorbay.opendataset.AnimalsWithAttributes2.loader.**AnimalsWithAttributes2**(*path: str*) → *tensorbay.dataset.dataset.Da*

Dataloader of the Animals with attributes 2 dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    classes.txt
    predicates.txt
    predicate-matrix-binary.txt
    JPEGImages/
        <classname>/
            <imagename>.jpg
        ...
    ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.BSTLD.loader

Dataloader of the BSTLD dataset.

tensorbay.opendataset.BSTLD.loader.**BSTLD**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the BSTLD dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    rgb/
        additional/
            2015-10-05-10-52-01_bag/
                <image_name>.jpg
                ...
            ...
        test/
            <image_name>.jpg
            ...
        train/
            2015-05-29-15-29-39_arastradero_traffic_light_loop_bag/
                <image_name>.jpg
                ...
            ...
    test.yaml
    train.yaml
    additional_train.yaml
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.CarConnection.loader

Dataloader of the The Car Connection Picture dataset.

tensorbay.opendataset.CarConnection.loader.**CarConnection**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the The Car Connection Picture dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    <imagename>.jpg
    ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.CoinImage.loader

Dataloader of the Coin Image dataset.

tensorbay.opendataset.CoinImage.loader.**CoinImage**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the Coin Image dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    classes.csv
    <imagename>.png
    ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.CompCars.loader

Dataloader of the CompCars dataset.

tensorbay.opendataset.CompCars.loader.**CompCars**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the CompCars dataset.

Parameters **path** – The root path of dataset. The file structure should be like:

```
<path>
    data/
        image/
            <make name id>/
                <model name id>/
                    <year>/
                        <image name>.jpg
                        ...
                    ...
                ...
            ...
        label/
            <make name id>/
                <model name id>/
                    <year>/
                        <image name>.txt
                        ...
                    ...
                ...
            ...
        misc/
            attributes.txt
            car_type.mat
            make_model_name.mat
        train_test_split/
            classification/
                train.txt
                test.txt
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.DeepRoute.loader

Dataloader of the DeepRoute dataset.

tensorbay.opendataset.DeepRoute.loader.**DeepRoute**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the DeepRoute dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    pointcloud/
        00001.bin
        00002.bin
        ...
        10000.bin
    groundtruth/
        00001.txt
        00002.txt
        ...
        10000.txt
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.DogsVsCats.loader

Dataloader of the DogsVsCats dataset.

tensorbay.opendataset.DogsVsCats.loader.**DogsVsCats**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the DogsVsCats dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    train/
        cat.0.jpg
        ...
        dog.0.jpg
        ...
    test/
        1000.jpg
        1001.jpg
        ...
```

Returns Loaded `Dataset` object.

tensorbay.opendataset.DownsampledImagenet.loader

Dataloader of the DownsampledImagenet dataset.

tensorbay.opendataset.DownsampledImagenet.loader.**DownsampledImagenet**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the DownsampledImagenet dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    valid_32x32/
        <imagename>.png
        ...
    valid_64x64/
        <imagename>.png
        ...
    train_32x32/
        <imagename>.png
        ...
    train_64x64/
        <imagename>.png
        ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.Elpv.loader

Dataloader of the Elpv dataset.

tensorbay.opendataset.Elpv.loader.**Elpv**(*path: str*) → *tensorbay.dataset.dataset.Dataset*
Dataloader of the Elpv dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    labels.csv
    images/
        cell0001.png
        ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.FLIC.loader

Dataloader of the FLIC dataset.

tensorbay.opendataset.FLIC.loader.**FLIC**(*path: str*) → *tensorbay.dataset.dataset.Dataset*
Dataloader of the FLIC dataset.

Parameters **path** – The root directory of the dataset. The folder structure should be like:

```
<path>
    exampls.mat
    images/
        2-fast-2-furious-00003571.jpg
        ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.FSDD.loader

Dataloader of the Free Spoken Digit dataset.

tensorbay.opendataset.FSDD.loader.**FSDD**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the Free Spoken Digit dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    recordings/
        0_george_0.wav
        0_george_1.wav
        ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.Flower.loader

Dataloader of the 17 Category Flower dataset and the 102 Category Flower dataset.

tensorbay.opendataset.Flower.loader.**Flower102**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the 102 Category Flower dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    jpg/
        image_00001.jpg
        ...
    imagelabels.mat
    setid.mat
```

Returns A loaded dataset.

tensorbay.opendataset.Flower.loader.**Flower17**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the 17 Category Flower dataset.

The dataset are 3 separate splits. The results in the paper are averaged over the 3 splits. We just use (trn1, val1, tst1) to split it.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    jpg/
        image_0001.jpg
        ...
    datasplits.mat
```

Returns A loaded dataset.

tensorbay.opendataset.HardHatWorkers.loader

Dataloader of the Hard Hat Workers dataset.

tensorbay.opendataset.HardHatWorkers.loader.**HardHatWorkers**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the Hard Hat Workers dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    annotations/
        hard_hat_workers0.xml
        ...
    images/
        hard_hat_workers0.png
        ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.HeadPoseImage.loader

Dataloader of the Head Pose Image dataset.

tensorbay.opendataset.HeadPoseImage.loader.**HeadPoseImage**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the Head Pose Image dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    Person01/
        person01100-90+0.jpg
        person01100-90+0.txt
        person01101-60-90.jpg
        person01101-60-90.txt
        ...
    Person02/
    Person03/
    ...
    Person15/
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.ImageEmotion.loader

Dataloader of the ImageEmotionAbstract dataset and the ImageEmotionArtphoto dataset.

tensorbay.opendataset.ImageEmotion.loader.**ImageEmotionAbstract**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the ImageEmotionAbstract dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    ABSTRACT_groundTruth.csv
```

(continues on next page)

(continued from previous page)

```
    abstract_xxxx.jpg
    ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.ImageEmotion.loader.**ImageEmotionArtphoto**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the ImageEmotionArtphoto dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    <filename>.jpg
    ...
```

Returns Loaded *Dataset* object

tensorbay.opendataset.JHU_CROWD.loader

Dataloader of the JHU-CROWD++ dataset.

tensorbay.opendataset.JHU_CROWD.loader.**JHU_CROWD**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the JHU-CROWD++ dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    train/
        images/
            0000.jpg
            ...
        gt/
            0000.txt
            ...
        image_labels.txt
    test/
    val/
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.KenyanFood.loader

Dataloader of the Kenyan Food or Nonfood dataset and Kenyan Food Type dataset.

tensorbay.opendataset.KenyanFood.loader.**KenyanFoodOrNonfood**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the Kenyan Food or Nonfood dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
        images/
            food/
                236171947206673742.jpg
```

(continues on next page)

(continued from previous page)

```
                ...
            nonfood/
                168223407.jpg
                ...
        data.csv
        split.py
        test.txt
        train.txt
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.KenyanFood.loader.**KenyanFoodType**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the Kenyan Food Type dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    test.csv
    test/
        bhaji/
            1611654056376059197.jpg
            ...
        chapati/
            1451497832469337023.jpg
            ...
        ...
    train/
        bhaji/
            190393222473009410.jpg
            ...
        chapati/
            1310641031297661755.jpg
            ...
    val/
        bhaji/
            1615408264598518873.jpg
            ...
        chapati/
            1553618479852020228.jpg
            ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.KylbergTexture.loader

Dataloader of the Kylberg Texture dataset.

tensorbay.opendataset.KylbergTexture.loader.**KylbergTexture**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the Kylberg Texture dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    originalPNG/
        <imagename>.png
```

(continues on next page)

(continued from previous page)

```
        ...
    withoutRotateAll/
        <imagename>.png
        ...
    RotateAll/
        <imagename>.png
        ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.LISATrafficLight.loader

Dataloader of the LISA traffic light dataset.

tensorbay.opendataset.LISATrafficLight.loader.**LISATrafficLight**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the LISA traffic light dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    Annotations/Annotations/
        daySequence1/
        daySequence2/
        dayTrain/
            dayClip1/
            dayClip10/
            ...
            dayClip9/
        nightSequence1/
        nightSequence2/
        nightTrain/
            nightClip1/
            nightClip2/
            ...
            nightClip5/
    daySequence1/daySequence1/
    daySequence2/daySequence2/
    dayTrain/dayTrain/
        dayClip1/
        dayClip10/
        ...
        dayClip9/
    nightSequence1/nightSequence1/
    nightSequence2/nightSequence2/
    nightTrain/nightTrain/
        nightClip1/
        nightClip2/
        ...
        nightClip5/
```

Returns Loaded *Dataset* object.

Raises **TypeError** – When frame number is discontinuous.

tensorbay.opendataset.LeedsSportsPose.loader

Dataloader of the LeedsSportsPose dataset.

tensorbay.opendataset.LeedsSportsPose.loader.**LeedsSportsPose**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the LeedsSportsPose dataset.

Parameters **path** – The root directory of the dataset. The folder structure should be like:

```
<path>
    joints.mat
    images/
        im0001.jpg
        im0002.jpg
        ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.NeolixOD.loader

Dataloader of the NeolixOD dataset.

tensorbay.opendataset.NeolixOD.loader.**NeolixOD**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the NeolixOD dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    bins/
        <id>.bin
    labels/
        <id>.txt
    ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.Newsgroups20.loader

Dataloader of the Newsgroups20 dataset.

tensorbay.opendataset.Newsgroups20.loader.**Newsgroups20**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the Newsgroups20 dataset.

Parameters **path** – The root directory of the dataset. The folder structure should be like:

```
<path>
    20news-18828/
        alt.atheism/
            49960
            51060
            51119
            51120
            ...
        comp.graphics/
```

(continues on next page)

(continued from previous page)

```
        comp.os.ms-windows.misc/
        comp.sys.ibm.pc.hardware/
        comp.sys.mac.hardware/
        comp.windows.x/
        misc.forsale/
        rec.autos/
        rec.motorcycles/
        rec.sport.baseball/
        rec.sport.hockey/
        sci.crypt/
        sci.electronics/
        sci.med/
        sci.space/
        soc.religion.christian/
        talk.politics.guns/
        talk.politics.mideast/
        talk.politics.misc/
        talk.religion.misc/
    20news-bydate-test/
    20news-bydate-train/
    20_newsgroups/
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.NightOwls.loader

Dataloader of the NightOwls dataset.

tensorbay.opendataset.NightOwls.loader.**NightOwls**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the NightOwls dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    nightowls_test/
        <image_name>.png
        ...
    nightowls_training/
        <image_name>.png
        ...
    nightowls_validation/
        <image_name>.png
        ...
    nightowls_training.json
    nightowls_validation.json
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.RP2K.loader

Dataloader of the RP2K dataset.

tensorbay.opendataset.RP2K.loader.**RP2K**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the RP2K dataset.

Parameters **path** – The root directory of the dataset. The file structure of RP2K looks like:

```
<path>
    all/
        test/
            <catagory>/
                <image_name>.jpg
                ...
            ...
        train/
            <catagory>/
                <image_name>.jpg
                ...
            ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.THCHS30.loader

Dataloader of the THCHS-30 dataset.

tensorbay.opendataset.THCHS30.loader.**THCHS30**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the THCHS-30 dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    lm_word/
        lexicon.txt
    data/
        A11_0.wav.trn
        ...
    dev/
        A11_101.wav
        ...
    train/
    test/
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.THUCNews.loader

Dataloader of the THUCNews dataset.

tensorbay.opendataset.THUCNews.loader.**THUCNews**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the THUCNews dataset.

Parameters **path** – The root directory of the dataset. The folder structure should be like:

```
<path>
    <category>/
        0.txt
        1.txt
        2.txt
        3.txt
        ...
    <category>/
    ...
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.TLR.loader

Dataloader of the TLR dataset.

tensorbay.opendataset.TLR.loader.**TLR**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the TLR dataset.

Parameters **path** – The root directory of the dataset. The file structure should like:

```
<path>
    root_path/
        Lara3D_URbanSeq1_JPG/
            frame_011149.jpg
            frame_011150.jpg
            frame_<frame_index>.jpg
            ...
        Lara_UrbanSeq1_GroundTruth_cvml.xml
```

Returns Loaded *Dataset* object.

tensorbay.opendataset.WIDER_FACE.loader

Dataloader of the WIDER FACE dataset.

tensorbay.opendataset.WIDER_FACE.loader.**WIDER_FACE**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the WIDER FACE dataset.

Parameters **path** – The root directory of the dataset. The file structure should be like:

```
<path>
    WIDER_train/
        images/
            0--Parade/
                0_Parade_marchingband_1_100.jpg
                0_Parade_marchingband_1_1015.jpg
```

(continues on next page)

(continued from previous page)

```
                0_Parade_marchingband_1_1030.jpg
                ...
            1--Handshaking/
            ...
            59--people--driving--car/
            61--Street_Battle/
    WIDER_val/
        ...
    WIDER_test/
        ...
    wider_face_split/
        wider_face_train_bbx_gt.txt
        wider_face_val_bbx_gt.txt
```

Returns Loaded *Dataset* object.

1.10.6 tensorbay.sensor

tensorbay.sensor.intrinsics

CameraMatrix, DistortionCoefficients and CameraIntrinsics.

CameraMatrix represents camera matrix. It describes the mapping of a pinhole camera model from 3D points in the world to 2D points in an image.

DistortionCoefficients represents camera distortion coefficients. It is the deviation from rectilinear projection including radial distortion and tangential distortion.

CameraIntrinsics represents camera intrinsics including camera matrix and distortion coeffecients. It describes the mapping of the scene in front of the camera to the pixels in the final image.

CameraMatrix, *DistortionCoefficients* and *CameraIntrinsics* class can all be initialized by `__init__()` or `loads()` method.

class tensorbay.sensor.intrinsics.**CameraIntrinsics**(*camera_matrix: Optional[Sequence[Sequence[float]]] = None*, ***, *_init_distortion: bool = True*, ***kwargs: float*)

Bases: *tensorbay.utility.repr.ReprMixin*

CameraIntrinsics represents camera intrinsics.

Camera intrinsic parameters including camera matrix and distortion coeffecients. They describe the mapping of the scene in front of the camera to the pixels in the final image.

Parameters

- **camera_matrix** – A 3x3 Sequence of the camera matrix.
- **_init_distortion** – Whether init distortion, default is True.
- ****kwargs** – Float values to initialize *CameraMatrix* and *DistortionCoefficients*.

_camera_matrix
A 3x3 Sequence of the camera matrix.

_distortion_coefficients
It is the deviation from rectilinear projection. It includes radial distortion and tangential distortion.

property camera_matrix
Get the camera matrix of the camera intrinsics.

Returns *CameraMatrix* class object containing fx, fy, cx, cy, skew(optional).

property distortion_coefficients
Get the distortion coefficients of the camera intrinsics, could be None.

Returns *DistortionCoefficients* class object containing tangential and radial distortion coefficients.

dumps() → Dict[str, Dict[str, float]]
Dumps the camera intrinsics into a dict.

Returns A dict containing camera intrinsics.

classmethod loads(*contents: Dict[str, Dict[str, float]]*) → _T
Loads CameraIntrinsics from a dict containing the information.

Parameters **contents** – A dict containig camera matrix and distortion coefficients.

Returns A *CameraIntrinsics* instance containing information from the contents dict.

project(*point: Sequence[float]*, *is_fisheye: bool = False*) → *tensorbay.geometry.vector.Vector2D*
Project a point to the pixel coordinates.

If distortion coefficients are provided, distort the point before projection.

Parameters

- **point** – A Sequence containing coordinates of the point to be projected.
- **is_fisheye** – Whether the sensor is fisheye camera, default is False.

Returns The coordinates on the pixel plane where the point is projected to.

set_camera_matrix(*matrix: Optional[Sequence[Sequence[float]]] = None*, ***kwargs: float*) → None
Set camera matrix of the camera intrinsics.

Parameters

- **matrix** – Camera matrix in 3x3 sequence.
- ****kwargs** – Contains fx, fy, cx, cy, skew(optional)

set_distortion_coefficients(***kwargs: float*) → None
Set distortion coefficients of the camera intrinsics.

Parameters ****kwargs** – Contains p1, p2, . . . , k1, k2, . . .

class tensorbay.sensor.intrinsics.**CameraMatrix**(*matrix: Optional[Sequence[Sequence[float]]] = None*, ***kwargs: float*)
Bases: *tensorbay.utility.repr.ReprMixin*

CameraMatrix represents camera matrix.

Camera matrix describes the mapping of a pinhole camera model from 3D points in the world to 2D points in an image.

Parameters

- **matrix** – A 3x3 Sequence of camera matrix.
- ****kwargs** – Float values with keys: “fx”, “fy”, “cx”, “cy” and “skew”(optional).

fx
The x axis focal length expressed in pixels.

fy
The y axis focal length expressed in pixels.

cx
The x coordinate of the so called principal point that should be in the center of the image.

cy
The y coordinate of the so called principal point that should be in the center of the image.

skew
It causes shear distortion in the projected image.

Raises **TypeError** – When only keyword arguments with incorrect keys are provided, or when no arguments are provided.

as_matrix() → numpy.ndarray
Return the camera matrix as a 3x3 numpy array.

Returns A 3x3 numpy array representing the camera matrix.

dumps() → Dict[str, float]
Dumps the camera matrix into a dict.

Returns A dict containing the information of the camera matrix.

classmethod **loads**(*contents: Dict[str, float]*) → _T
Loads CameraMatrix from a dict containing the information of the camera matrix.

Parameters **contents** – A dict containing the information of the camera matrix.

Returns A *CameraMatrix* instance contains the information from the contents dict.

project(*point: Sequence[float]*) → *tensorbay.geometry.vector.Vector2D*
Project a point to the pixel coordinates.

Parameters **point** – A Sequence containing the coordinates of the point to be projected.

Returns The pixel coordinates.

Raises **TypeError** – When the dimension of the input point is neither two nor three.

class tensorbay.sensor.intrinsics.**DistortionCoefficients**(***kwargs: float*)
Bases: *tensorbay.utility.repr.ReprMixin*

DistortionCoefficients represents camera distortion coefficients.

Distortion is the deviation from rectilinear projection including radial distortion and tangential distortion.

Parameters ****kwargs** – Float values with keys: k1, k2, ... and p1, p2, ...

Raises **TypeError** – When tangential and radial distortion is not provided to initialize class.

distort(*point: Sequence[float]*, *is_fisheye: bool = False*) → *tensorbay.geometry.vector.Vector2D*
Add distortion to a point.

Parameters

- **point** – A Sequence containing the coordinates of the point to be distorted.
- **is_fisheye** – Whether the sensor is fisheye camera, default is False.

Raises **TypeError** – When the dimension of the input point is neither two nor three.

Returns Distorted 2d point.

dumps() → Dict[str, float]
Dumps the distortion coefficients into a dict.

Returns A dict containing the information of distortion coefficients.

classmethod loads(*contents: Dict[str, float]*) → _T
Loads DistortionCoefficients from a dict containing the information.

Parameters **contents** – A dict containig distortion coefficients of a camera.

Returns A *DistortionCoefficients* instance containing information from the contents dict.

tensorbay.sensor.sensor

SensorType, Sensor, Lidar, Radar, Camera and FisheyeCamera.

SensorType is an enumeration type. It includes 'LIDAR', 'RADAR', 'CAMERA' and 'FISHEYE_CAMERA'.

Sensor defines the concept of sensor. It includes name, description, translation and rotation.

A *Sensor* class can be initialized by `Sensor.__init__()` or *Sensor.loads()* method.

Lidar defines the concept of lidar. It is a kind of sensor for measuring distances by illuminating the target with laser light and measuring the reflection.

Radar defines the concept of radar. It is a detection system that uses radio waves to determine the range, angle, or velocity of objects.

Camera defines the concept of camera. It includes name, description, translation, rotation, cameraMatrix and distortionCoefficients.

FisheyeCamera defines the concept of fisheye camera. It is an ultra wide-angle lens that produces strong visual distortion intended to create a wide panoramic or hemispherical image.

class tensorbay.sensor.sensor.**Camera**(*name: str*)
Bases: *tensorbay.utility.name.NameMixin*, *tensorbay.utility.type.TypeMixin*[*tensorbay.sensor.sensor.SensorType*]

Camera defines the concept of camera.

Camera includes name, description, translation, rotation, cameraMatrix and distortionCoefficients.

extrinsics
The translation and rotation of the camera.

Type *tensorbay.geometry.transform.Transform3D*

intrinsics
The camera matrix and distortion coefficients of the camera.

Type *tensorbay.sensor.intrinsics.CameraIntrinsics*

dumps() → Dict[str, Any]
Dumps the camera into a dict.

Returns A dict containing name, description, extrinsics and intrinsics.

classmethod loads(*contents: Dict[str, Any]*) → _T
Loads a Camera from a dict containing the camera information.

Parameters **contents** – A dict containing name, description, extrinsics and intrinsics.

Returns A *Camera* instance containing information from contents dict.

set_camera_matrix(*matrix: Optional[Sequence[Sequence[float]]] = None*, ***kwargs: float*) → None

Set camera matrix.

Parameters

- **matrix** – A 3x3 Sequence of camera matrix.
- ****kwargs** – Other float values to set camera matrix.

set_distortion_coefficients(***kwargs: float*) → None

Set distortion coefficients.

Parameters ****kwargs** – Float values to set distortion coefficients.

Raises **ValueError** – When intrinsics is not set yet.

class tensorbay.sensor.sensor.**FisheyeCamera**(*name: str*)

Bases: *tensorbay.utility.name.NameMixin*, *tensorbay.utility.type.TypeMixin*[*tensorbay.sensor.sensor.SensorType*]

FisheyeCamera defines the concept of fisheye camera.

Fisheye camera is an ultra wide-angle lens that produces strong visual distortion intended to create a wide panoramic or hemispherical image.

class tensorbay.sensor.sensor.**Lidar**(*name: str*)

Bases: *tensorbay.utility.name.NameMixin*, *tensorbay.utility.type.TypeMixin*[*tensorbay.sensor.sensor.SensorType*]

Lidar defines the concept of lidar.

Lidar is a kind of sensor for measuring distances by illuminating the target with laser light and measuring the reflection.

class tensorbay.sensor.sensor.**Radar**(*name: str*)

Bases: *tensorbay.utility.name.NameMixin*, *tensorbay.utility.type.TypeMixin*[*tensorbay.sensor.sensor.SensorType*]

Radar defines the concept of radar.

Radar is a detection system that uses radio waves to determine the range, angle, or velocity of objects.

class tensorbay.sensor.sensor.**Sensor**(*name: str*)

Bases: *tensorbay.utility.name.NameMixin*, *tensorbay.utility.type.TypeMixin*[*tensorbay.sensor.sensor.SensorType*]

Sensor defines the concept of sensor.

Sensor includes name, description, translation and rotation.

Parameters **name** – *Sensor*'s name.

Raises **TypeError** – Can not instantiate abstract class *Sensor*.

extrinsics

The translation and rotation of the sensor.

Type *tensorbay.geometry.transform.Transform3D*

dumps() → Dict[str, Any]

Dumps the sensor into a dict.

Returns A dict containing the information of the sensor.

static loads(*contents: Dict[str, Any]*) → _Type
Loads a Sensor from a dict containing the sensor information.

Parameters **contents** – A dict containing name, description and sensor extrinsics.

Returns A *Sensor* instance containing the information from the contents dict.

set_extrinsics(*transform: Union[None, tensorbay.geometry.transform.Transform3D, Sequence[Sequence[float]], numpy.ndarray] = None*, ***, *translation: Iterable[float] = (0, 0, 0)*, *rotation: Union[Iterable[float], quaternion.quaternion] = (1, 0, 0, 0)*) → None
Set the extrinsics of the sensor.

Parameters

- **transform** – A `Transform3D` object representing the extrinsics.
- **translation** – Translation parameters.
- **rotation** – Rotation in a sequence of [w, x, y, z] or 3x3 rotation matrix or numpy quaternion.

set_rotation(*rotation: Union[Iterable[float], quaternion.quaternion]*) → None
Set the rotation of the sensor.

Parameters **rotation** – Rotation in a sequence of [w, x, y, z] or numpy quaternion.

set_translation(*x: float*, *y: float*, *z: float*) → None
Set the translation of the sensor.

Parameters

- **x** – The x coordinate of the translation.
- **y** – The y coordinate of the translation.
- **z** – The z coordinate of the translation.

sensor.set_translation(x=1, y=2, z=3)

class tensorbay.sensor.sensor.**SensorType**(*value*)
Bases: *tensorbay.utility.type.TypeEnum*

SensorType is an enumeration type.

It includes 'LIDAR', 'RADAR', 'CAMERA' and 'FISHEYE_CAMERA'.

1.10.7 tensorbay.utility

tensorbay.utility.loads

tensorbay.utility.name

NameMixin, NameSortedDict, NameSortedList and NameOrderedDict.

NameMixin is a mixin class for instance which has immutable name and mutable description.

NameSortedDict is a sorted mapping class which contains *NameMixin*. The corrsponding key is the 'name' of *NameMixin*.

NameSortedList is a sorted sequence class which contains *NameMixin*. It is maintained in sorted order according to the 'name' of *NameMixin*.

NameOrderedDict is an ordered mapping class which contains *NameMixin*. The corrsponding key is the 'name' of *NameMixin*.

class tensorbay.utility.name.**NameMixin**(*name: str*, *description: Optional[str] = None*)
Bases: *tensorbay.utility.repr.ReprMixin*, tensorbay.utility.common.EqMixin

A mixin class for instance which has immutable name and mutable description.

Parameters

- **name** – Name of the class.
- **description** – Description of the class.

classmethod loads(*contents: Dict[str, str]*) → _P
Loads a NameMixin from a dict containing the information of the NameMixin.

Parameters **contents** – A dict containing the information of the *NameMixin*:

```
{
    "name": <str>
    "description": <str>
}
```

Returns A *NameMixin* instance containing the information from the contents dict.

property name
Return name of the instance.

Returns Name of the instance.

class tensorbay.utility.name.**NameOrderedDict**
Bases: *tensorbay.utility.user.UserMapping*[str, tensorbay.utility.name._T]

Name ordered dict is an ordered mapping which contains NameMixin.

The corrsponding key is the 'name' of *NameMixin*.

append(*value: _T*) → None
Store element in ordered dict.

Parameters **value** – *NameMixin* instance.

class tensorbay.utility.name.**NameSortedDict**(*data: Optional[Mapping[str, _T]] = None*)
Bases: *tensorbay.utility.user.UserMapping*[str, tensorbay.utility.name._T]

Name sorted dict keys are maintained in sorted order.

Name sorted dict is a sorted mapping which contains *NameMixin*. The corrsponding key is the 'name' of *NameMixin*.

Parameters **data** – A mapping from str to *NameMixin* which needs to be transferred to *NameSortedDict*.

add(*value: _T*) → None
Store element in name sorted dict.

Parameters **value** – *NameMixin* instance.

class tensorbay.utility.name.**NameSortedList**
Bases: Sequence[tensorbay.utility.name._T]

Name sorted list is a sorted sequence which contains NameMixin.

It is maintained in sorted order according to the 'name' of *NameMixin*.

add(*value: _T*) → None
Store element in name sorted list.

Parameters **value** – *NameMixin* instance.

get_from_name(*name: str*) → _T
Get element in name sorted list from name of NameMixin.

Parameters **name** – Name of *NameMixin* instance.

Returns The element to be get.

tensorbay.utility.repr

ReprType and ReprMixin.

ReprType is an enumeration type, which defines the repr strategy type and includes 'INSTANCE', 'SEQUENCE', 'MAPPING'.

ReprMixin provides customized repr config and method.

class tensorbay.utility.repr.**ReprMixin**
Bases: object

ReprMixin provides customized repr config and method.

class tensorbay.utility.repr.**ReprType**(*value*)
Bases: enum.Enum

ReprType is an enumeration type.

It defines the repr strategy type and includes 'INSTANCE', 'SEQUENCE' and 'MAPPING'.

tensorbay.utility.tbrn

TensorBay Resource Name (TBRN) related classes.

TBRNType is an enumeration type, which has 7 types: 'DATASET', 'SEGMENT', 'FRAME', 'SEGMENT_SENSOR', 'FRAME_SENSOR', 'NORMAL_FILE' and 'FUSION_FILE'.

TBRN is a TensorBay Resource Name(TBRN) parser and generator.

class tensorbay.utility.tbrn.**TBRN**(*dataset_name: Optional[str] = None*, *segment_name: Optional[str] = None*, *frame_index: Optional[int] = None*, *sensor_name: Optional[str] = None*, ***, *remote_path: Optional[str] = None*, *tbrn: Optional[str] = None*)
Bases: object

TBRN is a TensorBay Resource Name(TBRN) parser and generator.

Use as a generator:

```
>>> info = TBRN("VOC2010", "train", remote_path="2012_004330.jpg")
>>> info.type
<TBRNType.NORMAL_FILE: 5>
>>> info.get_tbrn()
'tb:VOC2010:train://2012_004330.jpg'
>>> print(info)
'tb:VOC2010:train://2012_004330.jpg'
```

Use as a parser:

```
>>> tbrn = "tb:VOC2010:train://2012_004330.jpg"
>>> info = TBRN(tbrn=tbrn)
>>> info.dataset
'VOC2010'
>>> info.segment_name
'train'
>>> info.remote_path
'2012_004330.jpg'
```

Parameters

- **dataset_name** – Name of the dataset.
- **segment_name** – Name of the segment.
- **frame_index** – Index of the frame.
- **sensor_name** – Name of the sensor.
- **remote_path** – Object path of the file.
- **tbrn** – Full TBRN string.

Raises **TypeError** – The TBRN is invalid.

property dataset_name
Return the dataset name.

Returns The dataset name.

property frame_index
Return the frame index.

Returns The frame index.

get_tbrn (*frame_width: int = 0*) → str
Generate the full TBRN string.

Parameters **frame_width** – Add '0' at the beginning of the frame_index, until it reaches the frame_width.

Returns The full TBRN string.

property remote_path
Return the object path.

Returns The object path.

property segment_name
Return the segment name.

Returns The segment name.

property sensor_name
Return the sensor name.

Returns The sensor name.

property type
Return the type of this TBRN.

Returns The type of this TBRN.

class tensorbay.utility.tbrn.**TBRNType**(*value*)

Bases: enum.Enum

TBRNType defines the type of a TBRN.

It has 7 types: 1. *TBRNType.DATASET*:

```
    "tb:VOC2012"

which means the dataset "VOC2012".
```

2. *TBRNType.SEGMENT*:

```
    "tb:VOC2010:train"

which means the "train" segment of dataset "VOC2012".
```

3. *TBRNType.FRAME*:

```
    "tb:KITTI:test:10"

which means the 10th frame of the "test" segment in dataset "KITTI".
```

4. *TBRNType.SEGMENT_SENSOR*:

```
    "tb:KITTI:test::lidar"

which means the sensor "lidar" of the "test" segment in dataset "KITTI".
```

5. *TBRNType.FRAME_SENSOR*:

```
    "tb:KITTI:test:10:lidar"

which means the sensor "lidar" which belongs to the 10th frame of the "test"␣
↪segment in
dataset "KITTI".

6. `TBRNType.NORMAL_FILE`::

"tb:VOC2012:train://2012_004330.jpg"

which means the file "2012_004330.jpg" of the "train" segment in normal␣
↪dataset "VOC2012".

7. `TBRNType.FUSION_FILE`::

"tb:KITTI:test:10:lidar://000024.bin"

which means the file "000024.bin" in fusion dataset "KITTI", its segment,␣
↪frame index and
sensor is "test", 10 and "lidar".
```

tensorbay.utility.type

TypeEnum, TypeMixin, TypeRegister and SubcatalogTypeRegister.

TypeEnum is a superclass for enumeration classes that need to create a mapping with class.

TypeMixin is a superclass for the class which needs to link with *TypeEnum*.

TypeRegister is a decorator, which is used for registering *TypeMixin* to *TypeEnum*.

SubcatalogTypeRegister is a decorator, which is used for registering *TypeMixin* to *TypeEnum*.

class tensorbay.utility.type.**SubcatalogTypeRegister**(*enum:* tensorbay.utility.type.TypeEnum)

Bases: object

SubcatalogTypeRegister is a decorator, which is used for registering TypeMixin to TypeEnum.

Parameters **enum** – The corresponding *TypeEnum* of the *TypeMixin*.

class tensorbay.utility.type.**TypeEnum**(*value*)

Bases: enum.Enum

TypeEnum is a superclass for enumeration classes that need to create a mapping with class.

The 'type' property is used for getting the corresponding class of the enumeration.

property type

Get the corresponding class.

Returns The corresponding class.

class tensorbay.utility.type.**TypeMixin**(**args*, ***kwds*)

Bases: Generic[tensorbay.utility.type._T]

TypeMixin is a superclass for the class which needs to link with TypeEnum.

It provides the class variable 'TYPE' to access the corresponding TypeEnum.

property enum

Get the corresponding TypeEnum.

Returns The corresponding TypeEnum.

class tensorbay.utility.type.**TypeRegister**(*enum:* tensorbay.utility.type.TypeEnum)

Bases: object

TypeRegister is a decorator, which is used for registering TypeMixin to TypeEnum.

Parameters **enum** – The corresponding *TypeEnum* of the *TypeMixin*.

tensorbay.utility.user

UserSequence, UserMutableSequence, UserMapping and UserMutableMapping.

UserSequence is a user-defined wrapper around sequence objects.

UserMutableSequence is a user-defined wrapper around mutable sequence objects.

UserMapping is a user-defined wrapper around mapping objects.

UserMutableMapping is a user-defined wrapper around mutable mapping objects.

class tensorbay.utility.user.**UserMapping**(**args*, ***kwds*)

Bases: Mapping[tensorbay.utility.user._K, tensorbay.utility.user._V], *tensorbay.utility.repr.ReprMixin*

UserMapping is a user-defined wrapper around mapping objects.

get(*key: _K*) → Optional[_V]
get(*key: _K*, *default: Union[_V, _T] = None*) → Union[_V, _T]

Return the value for the key if it is in the dict, else default.

Parameters

- **key** – The key for dict, which can be any immutable type.
- **default** – The value to be returned if key is not in the dict.

Returns The value for the key if it is in the dict, else default.

items() → AbstractSet[Tuple[_K, _V]]

Return a new view of the (key, value) pairs in dict.

Returns The (key, value) pairs in dict.

keys() → AbstractSet[_K]

Return a new view of the keys in dict.

Returns The keys in dict.

values() → ValuesView[_V]

Return a new view of the values in dict.

Returns The values in dict.

class tensorbay.utility.user.**UserMutableMapping**(**args*, ***kwds*)

Bases: *tensorbay.utility.user.UserMapping*[tensorbay.utility.user._K, tensorbay.utility.user._V], MutableMapping[tensorbay.utility.user._K, tensorbay.utility.user._V]

UserMutableMapping is a user-defined wrapper around mutable mapping objects.

clear() → None

Remove all items from the mutable mapping object.

pop(*key: _K*) → _V
pop(*key: _K*, *default: Union[_V, _T] = <object object>*) → Union[_V, _T]

Remove specified item and return the corresponding value.

Parameters

- **key** – The key for dict, which can be any immutable type.
- **default** – The value to be returned if the key is not in the dict and it is given.

Returns Value to be removed from the mutable mapping object.

popitem() → Tuple[_K, _V]

Remove and return a (key, value) pair as a tuple.

Pairs are returned in LIFO (last-in, first-out) order.

Returns A (key, value) pair as a tuple.

setdefault(*key: _K*, *default: Optional[_V] = None*) → _V

Set the value of the item with the specified key.

If the key is in the dict, return the corresponding value. If not, insert the key with a value of default and return default.

Parameters

- **key** – The key for dict, which can be any immutable type.
- **default** – The value to be set if the key is not in the dict.

Returns The value for key if it is in the dict, else default.

update(*__m: Mapping[_K, _V]*, ***kwargs: _V*) → None
update(*__m: Iterable[Tuple[_K, _V]]*, ***kwargs: _V*) → None
update(***kwargs: _V*) → None
Update the dict.

Parameters

- **__m** – A dict object, a generator object yielding a (key, value) pair or other object which has a *.keys()* method.
- ****kwargs** – The value to be added to the mutable mapping.

class tensorbay.utility.user.**UserMutableSequence**(**args*, ***kwds*)
Bases: MutableSequence[tensorbay.utility.user._T], *tensorbay.utility.repr.ReprMixin*

UserMutableSequence is a user-defined wrapper around mutable sequence objects.

append(*value: _T*) → None
Append object to the end of the mutable sequence.

Parameters **value** – Element to be appended to the mutable sequence.

clear() → None
Remove all items from the mutable sequence.

extend(*values: Iterable[_T]*) → None
Extend mutable sequence by appending elements from the iterable.

Parameters **values** – Elements to be Extended into the mutable sequence.

insert(*index: int*, *value: _T*) → None
Insert object before index.

Parameters

- **index** – Position of the mutable sequence.
- **value** – Element to be inserted into the mutable sequence.

pop(*index: int = - 1*) → _T
Return the item at index (default last) and remove it from the mutable sequence.

Parameters **index** – Position of the mutable sequence.

Returns Element to be removed from the mutable sequence.

remove(*value: _T*) → None
Remove the first occurrence of value.

Parameters **value** – Element to be removed from the mutable sequence.

reverse() → None
Reverse the items of the mutable sequence in place.

class tensorbay.utility.user.**UserSequence**(**args*, ***kwds*)

Bases: Sequence[tensorbay.utility.user._T], *tensorbay.utility.repr.ReprMixin*

UserSequence is a user-defined wrapper around sequence objects.

count(*value: _T*) → int

Return the number of occurrences of value.

Parameters **value** – The value to be counted the number of occurrences.

Returns The number of occurrences of value.

index(*value: _T*, *start: int = 0*, *stop: int = - 1*) → int

Return the first index of the value.

Parameters

- **value** – The value to be found.
- **start** – The start index of the subsequence.
- **stop** – The end index of the subsequence.

Returns The First index of value.

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z