TensorBay

Graviti

Dec 06, 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 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 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 Graviti Developer Tools 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 Instance

```
from tensorbay import GAS

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

Create a Dataset

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

List Dataset Names

```
dataset_names = 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, jobs=8)

# 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

dataset = Dataset("DatasetName", gas)
segment = dataset[0]

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

The following table lists a series of examples to help developers to use TensorBay(Table. 1.1).

Table 1.1: Examples

Examples	Description
Dogs vs Cats	Topic: Dataset Management Data Type: Image Label Type: *Classification*
20 Newsgroups	Topic: Dataset Management Data Type: Text Label Type: *Classification*
BSTLD	Topic: Dataset Management Data Type: Image Label Type: *Box2D*
Neolix OD	Topic: Dataset Management Data Type: Point Cloud Label Type: *Box3D*
Leeds Sports Pose	Topic: Dataset Management Data Type: Image Label Type: *Keypoints2D*
THCHS-30	Topic: Dataset Management Data Type: Audio Label Type: *Sentence*
VOC2012 Segmentation	Topic: Dataset Management Data Type: Image Label Types: *SemanticMask*, *InstanceMask*
Update Dataset	Topic: Update Dataset
Move And Copy	Topic: Move And Copy
Merge Datasets	Topic: Merge Datasets
Get Label Statistics	Topic: Get Label Statistics

1.2.1 Dogs vs Cats

This topic describes how to manage the Dogs vs Cats Dataset, which is a dataset with *Classification* label.

Authorize a Client Instance

An *accesskey* is needed to authenticate identity when using TensorBay.

```
from tensorbay import GAS

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

Create Dataset

```
gas.create_dataset("DogsVsCats")
```

Organize Dataset

Normally, `dataloader.py` and `catalog.json` are required to organize the "Dogs vs Cats" dataset into the `Dataset` instance. In this example, they are stored in the same directory like:

```
Dogs vs Cats/
    catalog.json
    dataloader.py
```

Step 1: Write the Catalog

A *catalog* contains all label information of one dataset, which is typically stored in a json file like `catalog.json`.

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

The only annotation type for "Dogs vs Cats" is *Classification*, and there are 2 *category* types.

Note: By passing the path of the `catalog.json`, `load_catalog()` supports loading the catalog into dataset.

Important: See *catalog table* for more catalogs with different label types.

Step 2: Write the Dataloader

A *dataloader* is needed to organize the dataset into a `Dataset` instance.

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

"""Dataloader of DogsVsCats dataset."""

import os

from tensorbay.dataset import Data, Dataset
from tensorbay.label import Classification
from tensorbay.opendataset._utility import glob

DATASET_NAME = "DogsVsCats"
_SEGMENTS = {"train": True, "test": False}


def DogsVsCats(path: str) -> Dataset:
    """`Dogs vs Cats <https://www.kaggle.com/c/dogs-vs-cats>`_ dataset.

    The file structure should be like::

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

    Arguments:
        path: The root directory of the dataset.

    Returns:
        Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.

    """
    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:
```

(continues on next page)

(continued from previous page)

```
            data = Data(image_path)
            if is_labeled:
                data.label.classification = Classification(os.path.basename(image_
↪path)[:3])
            segment.append(data)

    return dataset
```

See *Classification annotation* for more details.

There are already a number of dataloaders in TensorBay SDK provided by the community. Thus, instead of writing, importing an available dataloadert is also feasible.

```
from tensorbay.opendataset import DogsVsCats

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

Note: Note that catalogs are automatically loaded in available dataloaders, users do not have to write them again.

Important: See *dataloader table* for more examples of dataloaders with different label types.

Visualize Dataset

Optionally, the organized dataset can be visualized by **Pharos**, which is a TensorBay SDK plug-in. This step can help users to check whether the dataset is correctly organized. Please see *Visualization* for more details.

Upload Dataset

The organized "Dogs vs Cats" dataset can be uploaded to TensorBay for sharing, reuse, etc.

```
dataset_client = gas.upload_dataset(dataset, jobs=8)
dataset_client.commit("initial commit")
```

Similar with Git, the commit step after uploading can record changes to the dataset as a version. If needed, do the modifications and commit again. Please see *Version Control* for more details.

Read Dataset

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

```
dataset = Dataset("DogsVsCats", gas)
```

In *dataset* "Dogs vs Cats", there are two *segments*: `train` and `test`. Get the segment names by listing them all.

```
dataset.keys()
```

Get a segment by passing the required segment name.

```
segment = dataset["train"]
```

In the train *segment*, there is a sequence of *data*, which can be obtained by index.

```
data = segment[0]
```

In each *data*, there is a sequence of *Classification* annotations, which can be obtained 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 names in "categories" list of *catalog.json*. See *Classification* label format for more details.

Delete Dataset

```
gas.delete_dataset("DogsVsCats")
```

1.2.2 20 Newsgroups

This topic describes how to manage the 20 Newsgroups dataset, which is a dataset with *Classification* label type.

Authorize a Client Instance

An *accesskey* is needed to authenticate identity when using TensorBay.

```
from tensorbay import GAS

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

Create Dataset

```
gas.create_dataset("Newsgroups20")
```

Organize Dataset

Normally, `dataloader.py` and `catalog.json` are required to organize the "20 Newsgroups" dataset into the `Dataset` instance. In this example, they are stored in the same directory like:

```
20 Newsgroups/
    catalog.json
    dataloader.py
```

It takes the following steps to organize the "20 Newsgroups" dataset by the `Dataset` instance.

Step 1: Write the Catalog

A *Catalog* contains all label information of one dataset, which is typically stored in a json file like `catalog.json`.

```json
{
    "CLASSIFICATION": {
        "categories": [
            { "name": "alt.atheism" },
            { "name": "comp.graphics" },
            { "name": "comp.os.ms-windows.misc" },
            { "name": "comp.sys.ibm.pc.hardware" },
            { "name": "comp.sys.mac.hardware" },
            { "name": "comp.windows.x" },
            { "name": "misc.forsale" },
            { "name": "rec.autos" },
            { "name": "rec.motorcycles" },
            { "name": "rec.sport.baseball" },
            { "name": "rec.sport.hockey" },
            { "name": "sci.crypt" },
            { "name": "sci.electronics" },
            { "name": "sci.med" },
            { "name": "sci.space" },
            { "name": "soc.religion.christian" },
            { "name": "talk.politics.guns" },
            { "name": "talk.politics.mideast" },
            { "name": "talk.politics.misc" },
            { "name": "talk.religion.misc" }
        ]
    }
}
```

The only annotation type for "20 Newsgroups" is *Classification*, and there are 20 *category* types.

Note:

- The *categories* in *dataset* "20 Newsgroups" have parent-child relationship, and it use "." to sparate different levels.
- By passing the path of the `catalog.json`, *`load_catalog()`* supports loading the catalog into dataset.

Important: See *catalog table* for more catalogs with different label types.

Step 2: Write the Dataloader

A *dataloader* is neeeded to organize the dataset into a `Dataset` instance.

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

"""Dataloader of Newsgroups20 dataset."""

import os

from tensorbay.dataset import Data, Dataset
from tensorbay.label import Classification
from tensorbay.opendataset._utility import glob

DATASET_NAME = "Newsgroups20"
SEGMENT_DESCRIPTION_DICT = {
    "20_newsgroups": "Original 20 Newsgroups data set",
    "20news-bydate-train": (
        "Training set of the second version of 20 Newsgroups, "
        "which is sorted by date and has duplicates and some headers removed"
    ),
    "20news-bydate-test": (
        "Test set of the second version of 20 Newsgroups, "
        "which is sorted by date and has duplicates and some headers removed"
    ),
    "20news-18828": (
        "The third version of 20 Newsgroups, which has duplicates removed "
        "and includes only 'From' and 'Subject' headers"
    ),
}


def Newsgroups20(path: str) -> Dataset:
    """`20 Newsgroups <http://qwone.com/~jason/20Newsgroups/>`_ dataset.

    The folder structure should be like::

        <path>
            20news-18828/
                alt.atheism/
                    49960
                    51060
                    51119
                    51120
                    ...
                comp.graphics/
                comp.os.ms-windows.misc/
                comp.sys.ibm.pc.hardware/
                comp.sys.mac.hardware/
```

(continues on next page)

(continued from previous page)

```
                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/

    Arguments:
        path: The root directory of the dataset.

    Returns:
        Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.

    """
    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, segment_description in SEGMENT_DESCRIPTION_DICT.items():
        segment_path = os.path.join(root_path, segment_name)
        if not os.path.isdir(segment_path):
            continue

        segment = dataset.create_segment(segment_name)
        segment.description = segment_description

        text_paths = glob(os.path.join(segment_path, "*", "*"))
        for text_path in text_paths:
            category = os.path.basename(os.path.dirname(text_path))

            data = Data(
                text_path, target_remote_path=f"{category}/{os.path.basename(text_path)}.txt"
            )
            data.label.classification = Classification(category)
            segment.append(data)

    return dataset
```

See *Classification annotation* for more details.

Note: The data in "20 Newsgroups" do not have extensions so that a "txt" extension is added to the remote path of each data file to ensure the loaded dataset could function well on TensorBay.

There are already a number of dataloaders in TensorBay SDK provided by the community. Thus, instead of writing, importing an available dataloader is also feasible.

```
from tensorbay.opendataset import Newsgroups20

dataset = Newsgroups20("path/to/dataset/directory")
```

Note: Note that catalogs are automatically loaded in available dataloaders, users do not have to write them again.

Important: See *dataloader table* for dataloaders with different label types.

Visualize Dataset

Optionally, the organized dataset can be visualized by **Pharos**, which is a TensorBay SDK plug-in. This step can help users to check whether the dataset is correctly organized. Please see *Visualization* for more details.

Upload Dataset

The organized "20 Newsgroups" dataset can be uploaded to TensorBay for sharing, reuse, etc.

```
dataset_client = gas.upload_dataset(dataset, jobs=8)
dataset_client.commit("initial commit")
```

Similar with Git, the commit step after uploading can record changes to the dataset as a version. If needed, do the modifications and commit again. Please see *Version Control* for more details.

Read Dataset

Now "20 Newsgroups" dataset can be read from TensorBay.

```
dataset = Dataset("Newsgroups20", gas)
```

In *dataset* "20 Newsgroups", there are four *Segments*: `20news-18828`, `20news-bydate-test` and `20news-bydate-train`, `20_newsgroups`. Get the segment names by listing them all.

```
dataset.keys()
```

Get a segment by passing the required segment name.

```
segment = dataset["20news-18828"]
```

In the 20news-18828 *segment*, there is a sequence of *data*, which can be obtained by index.

```
data = segment[0]
```

In each *data*, there is a sequence of *Classification* annotations, which can be obtained by index.

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

There is only one label type in "20 Newsgroups" 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

```
gas.delete_dataset("Newsgroups20")
```

1.2.3 BSTLD

This topic describes how to manage the BSTLD Dataset, which is a dataset with *Box2D* label(Fig. 1.1).

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

Authorize a Client Instance

An *accesskey* is needed to authenticate identity when using TensorBay.

```
from tensorbay import GAS

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

Create Dataset

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

Organize Dataset

Normally, `dataloader.py` and `catalog.json` are required to organize the "BSTLD" dataset into the *Dataset* instance. In this example, they are stored in the same directory like:

```
BSTLD/
    catalog.json
    dataloader.py
```

Step 1: Write the Catalog

A *catalog* contains all label information of one dataset, which is typically stored in a json file like `catalog.json`.

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

The only annotation type for "BSTLD" is *Box2D*, and there are 13 *category* types and one *attributes* type.

Note: By passing the path of the `catalog.json`, *load_catalog()* supports loading the catalog into dataset.

Important: See *catalog table* for more catalogs with different label types.

Step 2: Write the Dataloader

A *dataloader* is needed to organize the dataset into a `Dataset` instance.

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

"""Dataloader of BSTLD dataset."""

import os

from tensorbay.dataset import Data, Dataset
from tensorbay.exception import ModuleImportError
from tensorbay.label import LabeledBox2D

DATASET_NAME = "BSTLD"

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


def BSTLD(path: str) -> Dataset:
    """`BSTLD <https://hci.iwr.uni-heidelberg.de/content\
    /bosch-small-traffic-lights-dataset>`_ 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

    Arguments:
```

(continues on next page)

(continued from previous page)

```
        path: The root directory of the dataset.

    Raises:
        ModuleImportError: When the module "yaml" can not be found.

    Returns:
        Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.

    """
    try:
        import yaml  # pylint: disable=import-outside-toplevel
    except ModuleNotFoundError as error:
        raise ModuleImportError(module_name=error.name, package_name="pyyaml") from error

    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
```

See *Box2D annotation* for more details.

There are already a number of dataloaders in TensorBay SDK provided by the community. Thus, instead of writing, importing an available dataloader is also feasible.

```
from tensorbay.opendataset import BSTLD

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

Note: Note that catalogs are automatically loaded in available dataloaders, users do not have to write them again.

Important: See *dataloader table* for dataloaders with different label types.

Visualize Dataset

Optionally, the organized dataset can be visualized by **Pharos**, which is a TensorBay SDK plug-in. This step can help users to check whether the dataset is correctly organized. Please see *Visualization* for more details.

Upload Dataset

The organized "BSTLD" dataset can be uploaded to TensorBay for sharing, reuse, etc.

```
dataset_client = gas.upload_dataset(dataset, jobs=8, skip_uploaded_files=True)
dataset_client.commit("initial commit")
```

Note: Set *skip_uploaded_files=True* to skip uploaded data. The data will be skiped if its name and segment name is the same as remote data.

Similar with Git, the commit step after uploading can record changes to the dataset as a version. If needed, do the modifications and commit again. Please see *Version Control* for more details.

Read Dataset

Now "BSTLD" dataset can be read from TensorBay.

```
dataset = Dataset("BSTLD", gas)
```

In *dataset* "BSTLD", there are three *segments*: `train`, `test` and `additional`. Get the segment names by listing them all.

```
dataset.keys()
```

Get a segment by passing the required segment name.

```
first_segment = dataset[0]
train_segment = dataset["train"]
```

In the train *segment*, there is a sequence of *data*, which can be obtained by index.

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

In each *data*, there is a sequence of *Box2D* annotations, which can be obtained 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 names in "categories" list of *catalog.json*. The information stored in *attributes* is one or several of the attributes in "attributes" list of *catalog.json*. See *Box2D* label format for more details.

Delete Dataset

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

1.2.4 Neolix OD

This topic describes how to manage the Neolix OD dataset, which is a dataset with *Box3D* label type (Fig. 1.2).

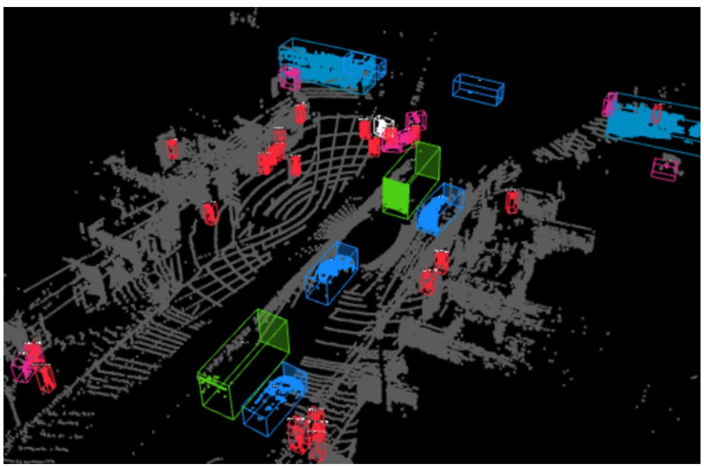

Fig. 1.2: The preview of a point cloud from "Neolix OD" with Box3D labels.

Authorize a Client Instance

An *accesskey* is needed to authenticate identity when using TensorBay.

```
from tensorbay import GAS

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

Create Dataset

```
gas.create_dataset("NeolixOD")
```

Organize Dataset

Normally, `dataloader.py` and `catalog.json` are required to organize the "Neolix OD" dataset into the *Dataset* instance. In this example, they are stored in the same directory like:

```
Neolix OD/
    catalog.json
    dataloader.py
```

Step 1: Write the Catalog

A *Catalog* contains all label information of one dataset, which is typically stored in a json file like `catalog.json`.

```
{
    "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" },
            { "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",
```

(continues on next page)

(continued from previous page)

```
                "description": "It indicates whether the object is truncated by the edge
↪of the image"
            }
        ]
    }
}
```

The only annotation type for “Neolix OD” is *Box3D*, and there are 15 *category* types and 3 *attributes* types.

Note: By passing the path of the `catalog.json`, *load_catalog()* supports loading the catalog into dataset.

Important: See *catalog table* for more catalogs with different label types.

Step 2: Write the Dataloader

A *dataloader* is needed to organize the dataset into a `Dataset` instance.

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

"""Dataloader of NeolixOD dataset."""

import os

from quaternion import from_rotation_vector

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

DATASET_NAME = "NeolixOD"


def NeolixOD(path: str) -> Dataset:
    """`Neolix OD <https://gas.graviti.cn/dataset\
    /graviti-open-dataset/NeolixOD>`_ dataset.

    The file structure should be like::

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

(continues on next page)

(continued from previous page)

```python

    Arguments:
        path: The root directory of the dataset.

    Returns:
        Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.

    """
    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(
                    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]))),
                    category=label_value[0],
                    attributes={
                        "Occlusion": int(label_value[1]),
                        "Truncation": bool(int(label_value[2])),
                        "Alpha": float(label_value[3]),
                    },
                )
                data.label.box3d.append(label)

        segment.append(data)
    return dataset
```

See *Box3D annotation* for more details.

There are already a number of dataloaders in TensorBay SDK provided by the community. Thus, instead of writing, importing an available dataloader is also feasible.

```
from tensorbay.opendataset import NeolixOD

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

Note: Note that catalogs are automatically loaded in available dataloaders, users do not have to write them again.

Important: See *dataloader table* for dataloaders with different label types.

Visualize Dataset

Optionally, the organized dataset can be visualized by **Pharos**, which is a TensorBay SDK plug-in. This step can help users to check whether the dataset is correctly organized. Please see *Visualization* for more details.

Upload Dataset

The organized "Neolix OD" dataset can be uploaded to tensorBay for sharing, reuse, etc.

```
dataset_client = gas.upload_dataset(dataset, jobs=8)
dataset_client.commit("initial commit")
```

Similar with Git, the commit step after uploading can record changes to the dataset as a version. If needed, do the modifications and commit again. Please see *Version Control* for more details.

Read Dataset

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

```
dataset = Dataset("NeolixOD", gas)
```

In *dataset* "Neolix OD", there is only one *segment*: `default`. Get a segment by passing the required segment name or the index.

```
segment = dataset[0]
```

In the default *segment*, there is a sequence of *data*, which can be obtained by index.

```
data = segment[0]
```

In each *data*, there is a sequence of *Box3D* annotations,

```
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 *Box3D* label format for more details.

Delete Dataset

```
gas.delete_dataset("NeolixOD")
```

1.2.5 Leeds Sports Pose

This topic describes how to manage the Leeds Sports Pose Dataset, which is a dataset with *Keypoints2D* label(Fig. 1.3).

Fig. 1.3: The preview of an image with labels from "Leeds Sports Pose".

Authorize a Client Instance

An *accesskey* is needed to authenticate identity when using TensorBay.

```
from tensorbay import GAS

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

Create Dataset

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

Organize Dataset

Normally, `dataloader.py` and `catalog.json` are required to organize the "Leeds Sports Pose" dataset into the `Dataset` instance. In this example, they are stored in the same directory like:

```
Leeds Sports Pose/
    catalog.json
    dataloader.py
```

Step 1: Write the Catalog

A *catalog* contains all label information of one dataset, which is typically stored in a json file like `catalog.json`.

```json
{
    "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]
                ],
```

(continues on next page)

(continued from previous page)

```
                "visible": "BINARY"
            }
        ]
    }
}
```

The only annotation type for “Leeds Sports Pose” is *Keypoints2D*.

Note: By passing the path of the `catalog.json`, *load_catalog()* supports loading the catalog into dataset.

Important: See *catalog table* for more catalogs with different label types.

Step 2: Write the Dataloader

A *dataloader* is needed to organize the dataset into a `Dataset` instance.

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

"""Dataloader of LeedsSportsPose dataset."""

import os

from tensorbay.dataset import Data, Dataset
from tensorbay.exception import ModuleImportError
from tensorbay.geometry import Keypoint2D
from tensorbay.label import LabeledKeypoints2D
from tensorbay.opendataset._utility import glob

DATASET_NAME = "LeedsSportsPose"


def LeedsSportsPose(path: str) -> Dataset:
    """`Leeds Sports Pose <http://sam.johnson.io/research/lsp.html>`_ dataset.

    The folder structure should be like::

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

    Arguments:
```

(continues on next page)

(continued from previous page)

```
        path: The root directory of the dataset.

    Raises:
        ModuleImportError: When the module "scipy" can not be found.

    Returns:
        Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.

    """
    try:
        from scipy.io import loadmat  # pylint: disable=import-outside-toplevel
    except ModuleNotFoundError as error:
        raise ModuleImportError(module_name=error.name) from error

    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"

        keypoints = LabeledKeypoints2D()
        for keypoint in joints[index]:
            keypoints.append(Keypoint2D(keypoint[0], keypoint[1], int(not keypoint[2])))

        data.label.keypoints2d.append(keypoints)
        segment.append(data)
    return dataset
```

See *Keipoints2D annotation* for more details.

There are already a number of dataloaders in TensorBay SDK provided by the community. Thus, instead of writing, importing an available dataloader is also feasible.

```
from tensorbay.opendataset import LeedsSportsPose

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

Note: Note that catalogs are automatically loaded in available dataloaders, users do not have to write them again.

Important: See *dataloader table* for dataloaders with different label types.

Visualize Dataset

Optionally, the organized dataset can be visualized by **Pharos**, which is a TensorBay SDK plug-in. This step can help users to check whether the dataset is correctly organized. Please see *Visualization* for more details.

Upload Dataset

The organized "BSTLD" dataset can be uploaded to TensorBay for sharing, reuse, etc.

```
dataset_client = gas.upload_dataset(dataset, jobs=8)
dataset_client.commit("initial commit")
```

Similar with Git, the commit step after uploading can record changes to the dataset as a version. If needed, do the modifications and commit again. Please see *Version Control* for more details.

Read Dataset

Now "Leeds Sports Pose" dataset can be read from TensorBay.

```
dataset = Dataset("LeedsSportsPose", gas)
```

In *dataset* "Leeds Sports Pose", there is one *segment* named `default`. Get it by passing the segment name or the index.

```
segment = dataset[0]
```

In the default *segment*, there is a sequence of *data*, which can be obtained by index.

```
data = segment[0]
```

In each *data*, there is a sequence of *Keypoints2D* annotations, which can be obtained 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 "Leeds Sports Pose" 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 *Keypoints2D* label format for more details.

Delete Dataset

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

1.2.6 THCHS-30

This topic describes how to manage the THCHS-30 Dataset, which is a dataset with *Sentence* label

Authorize a Client Instance

An *accesskey* is needed to authenticate identity when using TensorBay.

```
from tensorbay import GAS

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

Create Dataset

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

Organize Dataset

It takes the following steps to organize the "THCHS-30" dataset by the `Dataset` instance.

Step 1: Write the Catalog

A *Catalog* contains all label information of one dataset, which is typically stored in a json file. However the catalog of THCHS-30 is too large, instead of reading it from json file, we read it by mapping from subcatalog that is loaded by the raw file. Check the *dataloader* below for more details.

Important: See *catalog table* for more catalogs with different label types.

Step 2: Write the Dataloader

A *dataloader* is needed to organize the dataset into a `Dataset` instance.

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

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

import os
from itertools import islice
from typing import List

from tensorbay.dataset import Data, Dataset
from tensorbay.label import LabeledSentence, SentenceSubcatalog, Word
```

(continues on next page)

(continued from previous page)

```python
from tensorbay.opendataset._utility import glob

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


def THCHS30(path: str) -> Dataset:
    """`THCHS-30 <http://166.111.134.19:7777/data/thchs30/README.html>`_ dataset.

    The file structure should be like::

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

    Arguments:
        path: The root directory of the dataset.

    Returns:
        Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.

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

(continues on next page)

(continued from previous page)

```
        for line in islice(fp, 4, None):
            subcatalog.append_lexicon(line.strip().split())
    return subcatalog
```

See *Sentence annotation* for more details.

There are already a number of dataloaders in TensorBay SDK provided by the community. Thus, instead of writing, importing an available dataloadert is also feasible.

```
from tensorbay.opendataset import THCHS30

dataset = THCHS30("path/to/dataset/directory")
```

Note: Note that catalogs are automatically loaded in available dataloaders, users do not have to write them again.

Important: See *dataloader table* for dataloaders with different label types.

Visualize Dataset

Optionally, the organized dataset can be visualized by **Pharos**, which is a TensorBay SDK plug-in. This step can help users to check whether the dataset is correctly organized. Please see *Visualization* for more details.

Upload Dataset

The organized "THCHS-30" dataset can be uploaded to TensorBay for sharing, reuse, etc.

```
dataset_client = gas.upload_dataset(dataset, jobs=8)
dataset_client.commit("initial commit")
```

Similar with Git, the commit step after uploading can record changes to the dataset as a version. If needed, do the modifications and commit again. Please see *Version Control* for more details.

Read Dataset

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

```
dataset = Dataset("THCHS-30", gas)
```

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

```
dataset.keys()
```

Get a segment by passing the required segment name.

```
segment = dataset["dev"]
```

In the dev *segment*, there is a sequence of *data*, which can be obtained by index.

```
data = segment[0]
```

In each *data*, there is a sequence of *Sentence* annotations, which can be obtained 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 *Sentence* label format for more details.

Delete Dataset

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

1.2.7 VOC2012 Segmentation

This topic describes how to manage the VOC2012 Segmentation dataset, which is a dataset with *SemanticMask* and *InstanceMask* labels (Fig. 1.4 and Fig. 1.5).

Fig. 1.4: The preview of a semantic mask from “VOC2012 Segmentation”.

Fig. 1.5: The preview of a instance mask from “VOC2012 Segmentation”.

Authorize a Client Instance

An *accesskey* is needed to authenticate identity when using TensorBay.

```
from tensorbay import GAS

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

Create Dataset

```
gas.create_dataset("VOC2012Segmentation")
```

Organize Dataset

Normally, `dataloader.py` and `catalog.json` are required to organize the "VOC2012 Segmentation" dataset into the `Dataset` instance. In this example, they are stored in the same directory like:

```
VOC2012 Segmentation/
    catalog.json
    dataloader.py
```

It takes the following steps to organize "VOC2012 Segmentation" dataset by the `Dataset` instance.

Step 1: Write the Catalog

A *Catalog* contains all label information of one dataset, which is typically stored in a json file like `catalog.json`.

```
{
    "SEMANTIC_MASK": {
        "categories": [
            { "name": "background", "categoryId": 0 },
            { "name": "aeroplane", "categoryId": 1 },
            { "name": "bicycle", "categoryId": 2 },
            { "name": "bird", "categoryId": 3 },
            { "name": "boat", "categoryId": 4 },
            { "name": "bottle", "categoryId": 5 },
            { "name": "bus", "categoryId": 6 },
            { "name": "car", "categoryId": 7 },
            { "name": "cat", "categoryId": 8 },
            { "name": "chair", "categoryId": 9 },
            { "name": "cow", "categoryId": 10 },
            { "name": "diningtable", "categoryId": 11 },
            { "name": "dog", "categoryId": 12 },
            { "name": "horse", "categoryId": 13 },
            { "name": "motorbike", "categoryId": 14 },
            { "name": "person", "categoryId": 15 },
            { "name": "pottedplant", "categoryId": 16 },
            { "name": "sheep", "categoryId": 17 },
            { "name": "sofa", "categoryId": 18 },
```

(continues on next page)

(continued from previous page)

```
            { "name": "train", "categoryId": 19 },
            { "name": "tvmonitor", "categoryId": 20 },
            { "name": "void", "categoryId": 255 }
        ]
    },
    "INSTANCE_MASK": {
        "categories": [
            { "name": "background", "categoryId": 0 },
            { "name": "void", "categoryId": 255 }
        ]
    }
}
```

The annotation types for "VOC2012 Segmentation" are *SemanticMask* and *InstanceMask*, and there are 22 *category* types for *SemanticMask*. There are 2 *category* types for *InstanceMask*, category 0 represents the background, and category 255 represents the border of instances.

Note:

- By passing the path of the `catalog.json`, `load_catalog()` supports loading the catalog into dataset.
- The categories in *InstanceMaskSubcatalog* are for pixel values which are not instance ids.

Important: See *catalog table* for more catalogs with different label types.

Step 2: Write the Dataloader

A *dataloader* is needed to organize the dataset into a `Dataset` instance.

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

"""Dataloader of VOC2012Segmentation dataset."""

import os

from tensorbay.dataset import Data, Dataset
from tensorbay.label import InstanceMask, SemanticMask

_SEGMENT_NAMES = ("train", "val")
DATASET_NAME = "VOC2012Segmentation"


def VOC2012Segmentation(path: str) -> Dataset:
    """`VOC2012Segmentation <http://host.robots.ox.ac.uk/pascal/VOC/voc2012/>`_ dataset.

```

(continues on next page)

(continued from previous page)

```
    The file structure should be like::

        <path>/
            JPEGImages/
                <image_name>.jpg
                ...
            SegmentationClass/
                <mask_name>.png
                ...
            SegmentationObject/
                <mask_name>.png
                ...
            ImageSets/
                Segmentation/
                    train.txt
                    val.txt
                    ...
                ...
            ...

    Arguments:
        path: The root directory of the dataset.

    Returns:
        Loaded :class: `~tensorbay.dataset.dataset.Dataset` instance.

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

    image_path = os.path.join(root_path, "JPEGImages")
    semantic_mask_path = os.path.join(root_path, "SegmentationClass")
    instance_mask_path = os.path.join(root_path, "SegmentationObject")
    image_set_path = os.path.join(root_path, "ImageSets", "Segmentation")

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

    for segment_name in _SEGMENT_NAMES:
        segment = dataset.create_segment(segment_name)
        with open(os.path.join(image_set_path, f"{segment_name}.txt"), encoding="utf-8")␣
↪as fp:
            for stem in fp:
                stem = stem.strip()
                data = Data(os.path.join(image_path, f"{stem}.jpg"))
                label = data.label
                mask_filename = f"{stem}.png"
                label.semantic_mask = SemanticMask(os.path.join(semantic_mask_path, mask_
↪filename))
                label.instance_mask = InstanceMask(os.path.join(instance_mask_path, mask_
↪filename))

                segment.append(data)
```

(continues on next page)

(continued from previous page)

```
    return dataset
```

See *SemanticMask annotation* and *InstanceMask annotation* for more details.

There are already a number of dataloaders in TensorBay SDK provided by the community. Thus, instead of writing, importing an available dataloader is also feasible.

```
from tensorbay.opendataset import VOC2012Segmentation

dataset = VOC2012Segmentation("path/to/dataset/directory")
```

Note: Note that catalogs are automatically loaded in available dataloaders, users do not have to write them again.

Important: See *dataloader table* for dataloaders with different label types.

Upload Dataset

The organized "VOC2012 Segmentation" dataset can be uploaded to tensorBay for sharing, reuse, etc.

```
dataset_client = gas.upload_dataset(dataset, jobs=8)
dataset_client.commit("initial commit")
```

Similar with Git, the commit step after uploading can record changes to the dataset as a version. If needed, do the modifications and commit again. Please see *Version Control* for more details.

See the visualization on TensorBay website.

Read Dataset

Now "VOC2012 Segmentation" dataset can be read from TensorBay.

```
dataset = Dataset("VOC2012Segmentation", gas)
```

In *dataset* "VOC2012 Segmentation", there are two *segments*: `train` and `val`. Get a segment by passing the required segment name or the index.

```
segment_names = dataset.keys()
segment = dataset[0]
```

In the `train` *segment*, there is a sequence of *data*, which can be obtained by index.

```
data = segment[0]
```

In each *data*, there are one *SemanticMask* annotation and one *InstanceMask* annotation.

```
from PIL import Image

label_semantic_mask = data.label.semantic_mask
```

(continues on next page)

(continued from previous page)

```
semantic_all_attributes = label_semantic_mask.all_attributes
semantic_mask = Image.open(label_semantic_mask.open())
semantic_mask.show()

label_instance_mask = data.label.instance_mask
instance_all_attributes = label_instance_mask.all_attributes
instance_mask_url = label_instance_mask.get_url()
```

There are two label types in "VOC2012 Segmentation" dataset, which are `semantic_mask` and `instance_mask`. We can get the mask by `Image.open()` or get the mask url by `get_url()`. The information stored in *SemanticMask.all_attributes* is attributes for every category in `categories` list of `SEMANTIC_MASK`. The information stored in *InstanceMask.all_attributes* is attributes for every instance. See *SemanticMask* and *InstanceMask* label formats for more details.

Delete Dataset

```
gas.delete_dataset("VOC2012Segmentation")
```

1.2.8 Update Dataset

This topic describes how to update datasets, including:

- *Update Dataset Meta*
- *Update Dataset Notes*
- *Update Label*
- *Update Data*

The following scenario is used for demonstrating how to update data and label:

1. Upload a dataset.
2. Update the dataset's labels.
3. Add some data to the dataset.

Please see *Upload Dataset* for more information about the first step.
The last two steps will be introduced in detail.

Update Dataset Meta

TensorBay SDK supports a method to update dataset meta info.

```
gas.update_dataset("DATASET_NAME", alias="alias", is_public=True)
```

Update Dataset Notes

TensorBay SDK supports a method to update *dataset notes*. The dataset can be updated into continuous dataset by setting is_continuous to True.

```
dataset_client = gas.get_dataset("DATASET_NAME")
dataset_client.create_draft("draft-1")
dataset_client.update_notes(is_continuous=True)
dataset_client.commit("update notes")
```

Update Label

TensorBay SDK supports methods to update labels to overwrite previous labels.

Get a previously uploaded dataset and create a draft:

```
dataset_client.create_draft("draft-2")
```

Update the catalog if needed:

```
dataset_client.upload_catalog(dataset.catalog)
```

Overwrite previous labels with new label on dataset:

```
for segment in dataset:
    segment_client = dataset_client.get_segment(segment.name)
    for data in segment:
        segment_client.upload_label(data)
```

Commit the dataset:

```
dataset_client.commit("update labels")
```

Now dataset is committed with a version includes new labels.
Users can switch between different commits to use different version of labels.

Important: Uploading labels operation will overwrite all types of labels in data.

Update Data

Add new data to dataset.

```
gas.upload_dataset(dataset, jobs=8, skip_uploaded_files=True)
```

Set *skip_uploaded_files=True* to skip uploaded data.

Overwrite uploaded data to dataset.

```
gas.upload_dataset(dataset, jobs=8)
```

The default value of *skip_uploaded_files* is false, use it to overwrite uploaded data.

Note: The segment name and data name are used to identify data, which means if two data's segment names and data names are the same, then they will be regarded as one data.

Important: Uploading dataset operation will only add or overwrite data, Data uploaded before will not be deleted.

Delete segment by the segment name.

```
dataset_client.create_draft("draft-3")
dataset_client.delete_segment("SegmentName")
```

Delete data by the data remote path.

```
segment_client = dataset_client.get_segment("SegmentName")
segment_client.delete_data("a.png")
```

For a fusion dataset, TensorBay SDK supports deleting a frame by its id.

```
segment_client.delete_frame("00000000003W09TEMC1HXYMC74")
```

1.2.9 Move And Copy

This topic describes TensorBay dataset operations:

- *Copy Segment*
- *Move Segment*
- *Copy Data*
- *Move Data*

Take the Oxford-IIIT Pet as an example. Its structure looks like:

```
datasets/
    test/
        Abyssinian_002.jpg
        ...
    trainval/
        Abyssinian_001.jpg
        ...
```

Note: Before operating this dataset, fork it first.

Get the dataset client.

```
from tensorbay import GAS

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

(continues on next page)

(continued from previous page)

```
dataset_client = gas.get_dataset("OxfordIIITPet")
dataset_client.list_segment_names()
# test, trainval
```

There are currently two segments: `test` and `trainval`.

Copy Segment

Copy segment `test` to `test_1`.

```
dataset_client.create_draft("draft-1")
segment_client = dataset_client.copy_segment("test", "test_1")
segment_client.name
# test_1
dataset_client.list_segment_names()
# test, test_1, trainval
dataset_client.commit("copy test segment to test_1 segment")
```

Move Segment

Move segment `test` to `test_2`.

```
dataset_client.create_draft("draft-2")
segment_client = dataset_client.move_segment("test", "test_2")
segment_client.name
# test_2
dataset_client.list_segment_names()
# test_1, trainval, test_2
dataset_client.commit("move test segment to test_2 segment")
```

Copy Data

Copy all data with prefix `Abyssinian` in both `test_1` and `trainval` segments to `abyssinian` segment.

```
dataset_client.create_draft("draft-3")
target_segment_client = dataset_client.create_segment("abyssinian")
for name in ["test_1", "trainval"]:
    segment_client = dataset_client.get_segment(name)
    copy_files = []
    for file_name in segment_client.list_data_paths():
        if file_name.startswith("Aabyssinian"):
            copy_files.append(file_name)
    target_segment_client.copy_data(copy_files, source_client=segment_client)

dataset_client.list_segment_names()
# test_1, test_2, trainval, abyssinian
dataset_client.commit("add abyssinian segment")
```

Move Data

Split `trainval` segment into `train` and `val`:

1. Extract 500 data from `trainval` to `val` segment.
2. Move `trainval` to `train`.

```
import random

dataset_client.create_draft("draft-4")
val_segment_client = dataset_client.create_segment("val")
trainval_segment_client = dataset_client.get_segment("trainval")

# list_data_paths will return a lazy list, get and delete data are not supports at one␣
↪time.
data_paths = list(trainval_segment_client.list_data_paths())

# Generate 500 random numbers.
val_random_numbers = random.sample(range(0, len(data_paths)), 500)

# Get the data path list by random index list.
val_ramdom_paths = [data_paths[index] for index in val_random_numbers]

# Move all data of the val random path list from trainval to train segment
val_segment_client.move_data(val_ramdom_paths, source_client=trainval_segment_client)
dataset_client.move_segment("trainval", "train")

dataset_client.list_segment_names()
# train, val, test_1, test_2, abyssinian
dataset_client.commit("split train and val segment")
```

Note: The data storage space will only be calculated once when a segment is copied.

Note: TensorBay SDK supports three strategies to solve the conflict when the target segment/data already exists, which can be set as an keyword argument in the above-mentioned functions.

- abort(default): abort the process by raising InternalServerError.
- skip: skip moving or copying segment/data.
- override: override the whole target segment/data with the source segment/data.

1.2.10 Merge Datasets

This topic describes the merge dataset operation.

Take the Oxford-IIIT Pet and Dogs vs Cats as examples. Their structures looks like:

```
Oxford-IIIT Pet/
    test/
        Abyssinian_002.jpg
        ...
    trainval/
        Abyssinian_001.jpg
        ...

Dogs vs Cats/
    test/
        1.jpg
        10.jpg
        ...
    train/
        cat.0.jpg
        cat.1.jpg
        ...
```

There are lots of pictures of cats and dogs in these two datasets, merge them to get a more diverse dataset.

Note: Before merging datasets, fork both of the open datasets first.

Create a dataset which is named `mergedDataset`.

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
dataset_client = gas.create_dataset("mergedDataset")
dataset_client.create_draft("merge dataset")
```

Copy all segments in `OxfordIIITPetDog` to `mergedDataset`.

```
pet_dataset_client = gas.get_dataset("OxfordIIITPet")
dataset_client.copy_segment("train", target_name="trainval", source_client=pet_dataset_
↪client)
dataset_client.copy_segment("test", source_client=pet_dataset_client)
```

Use the catalog of OxfordIIITPet as the catalog of the merged dataset.

```
dataset_client.upload_catalog(pet_dataset_client.get_catalog())
```

Unify categories of `train` segment.

```
from tensorbay.dataset import Data

segment_client = dataset_client.get_segment("train")
for remote_data in segment_client.list_data():
```

(continues on next page)

(continued from previous page)

```
    data = Data(remote_data.path)
    data.label = remote_data.label
    data.label.classification.category = data.label.classification.category.split(".")[0]
    segment_client.upload_label(data)
```

Note: The category in `OxfordIIITPet` is of two-level formats, like `cat.Abyssinian`, but in `Dogs vs Cats` it only has one level, like `cat`. Thus it is important to unify the categories, for example, rename `cat.Abyssinian` to `cat`.

Copy data from `Dogs vs Cats` to `mergedDataset`.

```
pet_dataset_client = gas.get_dataset("DogsVsCats")
for name in ["test", "train"]:
    source_segment_client = pet_dataset_client.get_segment(name)
    segment_client = dataset_client.get_segment(name)
    segment_client.copy_data(
        source_segment_client.list_data_paths(), source_client=source_segment_client
    )
```

1.2.11 Get Label Statistics

This topic describes the get label statistics operation.

Label statistics of dataset could be obtained via *get_label_statistics()* as follows:

```
>>> from tensorbay import GAS
>>> ACCESS_KEY = "Accesskey-*****"
>>> gas = GAS(ACCESS_KEY)
>>> dataset_client = gas.get_dataset("targetDataset")
>>> statistics = dataset_client.get_label_statistics()
>>> statistics
Statistics {
    'BOX2D': {...},
    'BOX3D': {...},
    'KEYPOINTS2D': {...}
}
```

The details of the statistics structure for the targetDataset are as follows:

```
{
    "BOX2D": {
        "quantity": 1508722,
        "categories": [
            {
                "name": "vehicle.bike",
                "quantity": 8425,
                "attributes": [
                    {
                        "name": "trafficLightColor",
                        "enum": ["none", "red", "yellow"],
                        "quantities": [8420, 3, 2]
```

(continues on next page)

(continued from previous page)

```
                    }
                ]
            }
        ],
        "attributes": [
            {
                "name": "trafficLightColor",
                "enum": ["none", "red", "yellow", "green"],
                "quantities": [1356224, 54481, 4107, 93910]
            }
        ]
    },
    "BOX3D": {
        "quantity": 1234
    },
    "KEYPOINTS2D":{
        "quantity": 43234,
        "categories":[
            {
                "name": "person.person",
                "quantity": 43234
            }
        ]
    }
}
```

Note: The method *dumps()* of *Statistics* can dump the statistics into a dict.

1.3 Dataset Management

This topic describes dataset management, including:

- *Organize Dataset*
- *Upload Dataset*
- *Read Dataset*
- *Update Dataset*
- *Move and Copy*
- *Merge Datasets*
- *Get Label Statistics*

1.3.1 Organize Dataset

TensorBay SDK supports methods to organize local datasets into uniform TensorBay *dataset structure*. The typical steps to organize a local dataset:

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

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

Take the *Organization of BSTLD* as an example.

1.3.2 Upload Dataset

For an organized local dataset (i.e. the initialized `Dataset` instance), users can:

- Upload it to TensorBay.
- Read it directly.

This section mainly discusses the uploading operation. There are plenty of benefits of uploading local datasets to TensorBay.

- **REUSE**: uploaded datasets can be reused without preprocessing again.
- **SHARING**: uploaded datasets can be shared the with your team or the community.
- **VISUALIZATION**: uploaded datasets can be visualized without coding.
- **VERSION CONTROL**: different versions of one dataset can be uploaded and controlled conveniently.

Note: During uploading dataset or data, if the remote path of the data is the same as another data under the same segment, the old data will be replaced.

Take the *Upload Dataset of BSTLD* as an example.

1.3.3 Read Dataset

Two types of datasets can be read from TensorBay:

- Datasets uploaded by yourself as mentioned in *Upload Dataset*.
- Datasets uploaded by the shared Open Datasets platform.

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

Note: Visit my datasets(or team datasets) panel of TensorBay platform to check all datasets that can be read.

Take the *Read Dataset of BSTLD* as an example.

1.3.4 Update Dataset

Since TensorBay supports version control, users can update dataset meta, notes, data and labels to a new commit of a dataset. Thus, different versions of data and labels can coexist in one dataset, which greatly facilitates the datasets' maintenance.

Please see *Update dataset* example for more details.

1.3.5 Move and Copy

TensorBay supports four methods to copy or move data in datasets:

- copy segments
- copy data
- move segments
- move data

Copy is supported within a dataset or between datasets.

Moving is only supported within one dataset.

Note: The target dataset of copying and moving must be in *draft* status.

Please see *Move and copy* example for more details.

1.3.6 Merge Datasets

Since TensorBay supports copy operation between different datasets, users can use it to merge datasets.

Please see *Merge Datasets* example for more details.

1.3.7 Get Label Statistics

TensorBay supports getting label statistics of dataset.

Please see *Get Label Statistics* example for more details.

1.4 Version Control

TensorBay supports dataset version control. There can be multiple versions in one dataset.

1.4.1 Getting Started with Version Control

Commit

The basic element of TensorBay version control system is *commit*. Each commit of a TensorBay dataset is a **read-only** version. Take the VersionControlDemo Dataset as an example.

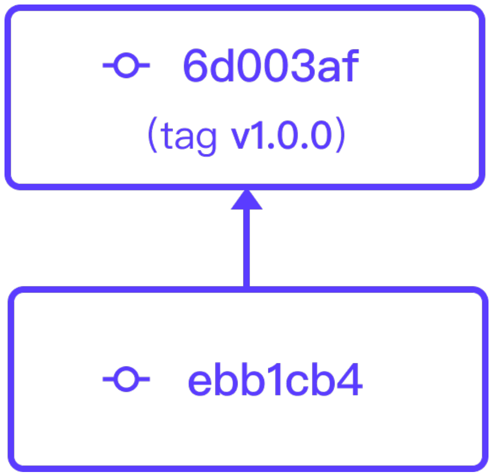

Fig. 1.6: The first two commits of dataset "VersionControlDemo".

Note: "VersionControlDemo" is an open dataset on Graviti Open Datasets platform, Please fork it before running the following demo code.

At the very beginning, there are only two commits in this dataset(Fig. 1.6). The code below checkouts to the first commit and check the data amount.

```
from tensorbay import GAS
from tensorbay.dataset import Dataset

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
commits = dataset_client.list_commits()

FIRST_COMMIT_ID = "ebb1cb46b36f4a4b922a40fb01574517"
version_control_demo = Dataset("VersionControlDemo", gas, revision=FIRST_COMMIT_ID)
train_segment = version_control_demo["train"]
print(f"data amount: {len(train_segment)}.")
# data amount: 4.
```

As shown above, there are 4 data in the train segment.

The code below checkouts to the second commit and check the data amount.

```
SECOND_COMMIT_ID = "6d003af913564943a83d705ff8440298"
version_control_demo = Dataset("VersionControlDemo", gas, revision=SECOND_COMMIT_ID)
train_segment = version_control_demo["train"]
print(f"data amount: {len(train_segment)}.")
# data amount: 8.
```

As shown above, there are 8 data in the train segment.

See *Draft and Commit* for more details about commit.

Draft

So how to create a dataset with multiple commits? A commit comes from a *draft*, which is a concept that represents a **writable** workspace.

Typical steps to create a new commit:

- Create a draft.
- Do the modifications/update in this draft.
- Commit this draft into a commit.

Note that the first "commit" occurred in the third step above is a verb. It means the action to turn a draft into a commit.

Figure. 1.7 demonstrates the relations between drafts and commits.

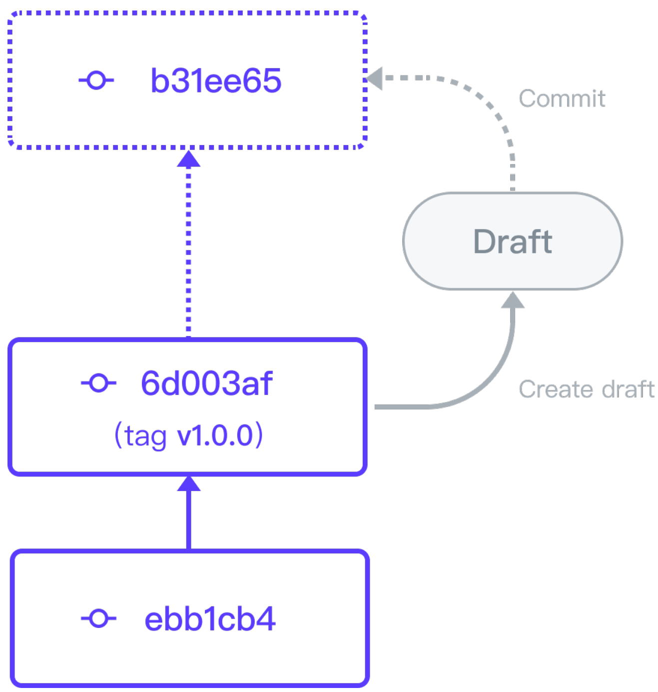

Fig. 1.7: The relations between a draft and commits.

The following code block creates a draft, adds a new segment to the "VersionControlDemo" dataset and does the commit operation.

```
import os
from tensorbay.dataset import Segment

TEST_IMAGES_PATH = "path/to/test_images"

dataset_client = gas.get_dataset("VersionControlDemo")
dataset_client.create_draft("draft-1")

```

(continues on next page)

(continued from previous page)

```
test_segment = Segment("test")

for image_name in os.listdir(TEST_IMAGES_PATH):
    data = Data(os.path.join(TEST_IMAGES_PATH, image_name))
    test_segment.append(data)

dataset_client.upload_segment(test_segment, jobs=8)
dataset_client.commit("add test segment")
```

See *Draft and Commit* for more details about draft.

Tag

For the convenience of marking major commits and switching between different commits, TensorBay provides the *tag* concept. The typical usage of tag is to mark released versions of a dataset.

The tag "v1.0.0" in Fig. 1.6 is added by

```
dataset_client.create_tag("v1.0.0", revision=SECOND_COMMIT_ID)
```

See *Tag* for more details about tag.

Branch

Sometimes, users may need to create drafts upon an early (not the latest) commit. For example, in an algorithm team, each team member may do modifications/update based on different versions of the dataset. This means a commit list may turn into a commit tree.

For the convenience of maintaining a commit tree, TensorBay provides the *branch* concept.

Actually, the commit list (Fig. 1.6) above is the default branch named "main".

The code block below creates a branch "with-label" based on the *revision* "v1.0.0", and adds *classification* label to the "train" segment.

Figure. 1.8 demonstrates the two branches.

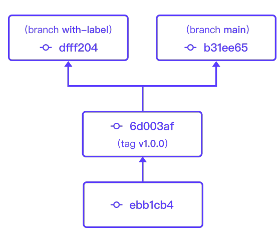

Fig. 1.8: The relations between branches.

```
from tensorbay.label import Catalog, Classification, ClassificationSubcatalog

TRAIN_IMAGES_PATH = "path/to/train/images"

catalog = Catalog()
classification_subcatalog = ClassificationSubcatalog()
```

(continues on next page)

(continued from previous page)

```
classification_subcatalog.add_category("zebra")
classification_subcatalog.add_category("horse")
catalog.classification = classification_subcatalog

dataset_client.upload_catalog(catalog)
dataset_client.create_branch("with-label", revision="v1.0.0")
dataset_client.create_draft("draft-2")

train_segment = Segment("train")
train_segment_client = dataset_client.get_segment(train_segment.name)

for image_name in os.listdir("path/to/train_images/"):
    data = Data(os.path.join(TRAIN_IMAGES_PATH, image_name))
    data.label.classification = Classification(image_name[:5])
    train_segment.append(data)
    train_segment_client.upload_label(data)

dataset_client.commit("add labels to train segment")
```

See *Branch* for more details about branch.

1.4.2 Draft and Commit

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

Similar with Git, a *commit* is a version of a dataset, which contains the changes compared with the former commit.

Unlike Git, a *draft* is a new concept which represents a workspace in which changing the dataset is allowed.

In TensorBay SDK, the dataset client supplies the function of version control.

Authorization

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
dataset_client = gas.create_dataset("DatasetName")
```

Create Draft

TensorBay SDK supports creating the draft straightforwardly, which is based on the current branch. Note that currently there can be only one open draft in each branch.

```
dataset_client.create_draft("draft-1")
```

Then the dataset client will change the status to “draft” and store the draft number. The draft number will be auto-increasing every time a draft is created.

```
is_draft = dataset_client.status.is_draft
# is_draft = True (True for draft, False for commit)
draft_number = dataset_client.status.draft_number
# draft_number = 1
branch_name = dataset_client.status.branch_name
# branch_name = main
```

Also, TensorBay SDK supports creating a draft based on a given branch.

```
dataset_client.create_draft("draft-1", branch_name="main")
```

List Drafts

The draft number can be found through listing drafts.

`status` includes "OPEN", "CLOSED", "COMMITTED" and None where None means listing drafts in all status. `branch_name` refers to the branch name of the draft to be listed.

```
drafts = dataset_client.list_drafts(status="CLOSED", branch_name="branch-1")
```

Get Draft

```
draft = dataset_client.get_draft(draft_number=1)
```

Commit Draft

After the commit, the draft will be closed.

```
dataset_client.commit("commit-1", "commit description")
is_draft = dataset_client.status.is_draft
# is_draft = False (True for draft, False for commit)
commit_id = dataset_client.status.commit_id
# commit_id = "***"
```

Get Commit

```
commit = dataset_client.get_commit(commit_id)
```

List Commits

```
commits = dataset_client.list_commits()
```

Checkout

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

Note: Here, *revision* is the information to locate the specific commit, which can be the commit id, the branch, or the tag.

1.4.3 Branch

TensorBay supports diverging from the main line of development and continue to do work without messing with that main line. Like Git, the way Tensorbay branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Tensorbay encourages workflows that branch often, even multiple times in a day.

Before operating branches, a dataset client instance with existing commit is needed.

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
dataset_client = gas.create_dataset("DatasetName")
dataset_client.create_draft("draft-1")
# Add some data to the dataset.
dataset_client.commit("commit-1", tag="V1")
commit_id_1 = dataset_client.status.commit_id

dataset_client.create_draft("draft-2")
# Do some modifications to the dataset.
dataset_client.commit("commit-2", tag="V2")
commit_id_2 = dataset_client.status.commit_id
```

Create Branch

Create Branch on the Current Commit

TensorBay SDK supports creating the branch straightforwardly, which is based on the current commit.

```
dataset_client.create_branch("T123")
```

Then the dataset client will storage the branch name. "main" is the default branch, it will be created when init the dataset

```
branch_name = dataset_client.status.branch_name
# branch_name = "T123"
commit_id = dataset_client.status.commit_id
# commit_id = "xxx"
```

Create Branch on a Revision

Also, creating a branch based on a revision is allowed.

```
dataset_client.create_branch("T123", revision=commit_id_2)
dataset_client.create_branch("T123", revision="V2")
dataset_client.create_branch("T123", revision="main")
```

The dataset client will checkout to the branch. The stored commit id is from the commit which the branch points to.

```
branch_name = dataset_client.status.branch_name
# branch_name = "T123"
commit_id = dataset_client.status.commit_id
# commit_id = "xxx"
```

Specially, creating a branch based on a former commit is permitted.

```
dataset_client.create_branch("T1234", revision=commit_id_1)
dataset_client.create_branch("T1234", revision="V1")
```

Similarly, the dataset client will checkout to the branch.

```
branch_name = dataset_client.status.branch_name
# branch_name = "T1234"
commit_id = dataset_client.status.commit_id
# commit_id = "xxx"
```

Then, through creating and committing the draft based on the branch, diverging from the current line of development can be realized.

```
dataset_client.create_draft("draft-3")
# Do some modifications to the dataset.
dataset_client.commit("commit-3", tag="V3")
```

List Branches

```
branches = dataset_client.list_branches()
```

Get Branch

```
branch = dataset_client.get_branch("T123")
```

Delete Branch

```
dataset_client.delete_branch("T123")
```

1.4.4 Tag

TensorBay supports tagging specific commits in a dataset's history as being important. Typically, people use this functionality to mark release revisions (v1.0, v2.0 and so on).

Before operating tags, a dataset client instance with existing commit is needed.

```
from tensorbay import GAS

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
dataset_client = gas.create_dataset("DatasetName")
dataset_client.create_draft("draft-1")
# do the modifications in this draft
```

Create Tag

TensorBay SDK supports three approaches of creating the tag.

First is to create the tag when committing.

```
dataset_client.commit("commit-1", tag="Tag-1")
```

Second is to create the tag straightforwardly, which is based on the current commit.

```
dataset_client.create_tag("Tag-1")
```

Third is to create tag on an existing commit.

```
commit_id = dataset_client.status.commit_id
dataset_client.create_tag("Tag-1", revision=commit_id)
```

Get Tag

```
tag = dataset_client.get_tag("Tag-1")
```

List Tags

```
tags = dataset_client.list_tags()
```

Delete Tag

```
dataset_client.delete_tag("Tag-1")
```

1.4.5 Diff

TensorBay supports showing changes between commits or drafts.

Before operating the *diff*, a dataset client instance with commits is needed. See more details in *Draft and Commit*

Get Diff

TensorBay SDK allows getting the dataset *diff* through *basehead*. Currently, only obtaining the *diff* between the head and its parent commit is supported; that is, the head is the given version(commit or draft) while the base is parent commit of the head.

```
diff = dataset_client.get_diff(head=head)
```

The type of the head indicates the version status: `string` for commit, `int` for draft.

Get Diff on Revision

For example, the following diff records the difference between the commit whose id is `"3bc35d806e0347d08fc23564b82737dc"` and its parent commit.

```
diff = dataset_client.get_diff(head="3bc35d806e0347d08fc23564b82737dc")
```

Get Diff on Draft Number

For example, the following diff records the difference between the draft whose draft number is 1 and its parent commit.

```
diff = dataset_client.get_diff(head=1)
```

Diff Object

The structure of the returning `DatasetDiff` looks like:

```
dataset_diff
├── segment_diff
│   ├── action
│   │   └── <str>
│   ├── data_diff
│   │   ├── file_diff
│   │   │   └── action
│   │   │       └── <str>
│   │   └── label_diff
│   │       └── action
│   │           └── <str>
│   │
```

(continues on next page)

(continued from previous page)

```
│   └── ...
├── segment_diff
│   ├── action
│   │   └── <str>
│   ├── data_diff
│   │   ├── file_diff
│   │   │   └── action
│   │   │       └── <str>
│   │   └── label_diff
│   │       └── action
│   │           └── <str>
│   └── ...
└── ...
```

The *DatasetDiff* is a list which is composed of *SegmentDiff* recording the changes of the segment. The *SegmentDiff* is a lazy-load sequence which is composed of *DataDiff* recording the changes of data.

The attribute “action” represents the status difference of the relative resource. It is an enum which includes:

- unmodify
- add
- delete
- modify

1.4.6 Squash and Merge

TensorBay supports squashing and merging between different branches.

Before *squash_and_merge()*, a dataset client instance with commits on different branches is needed. See more details in *Draft and Commit*.

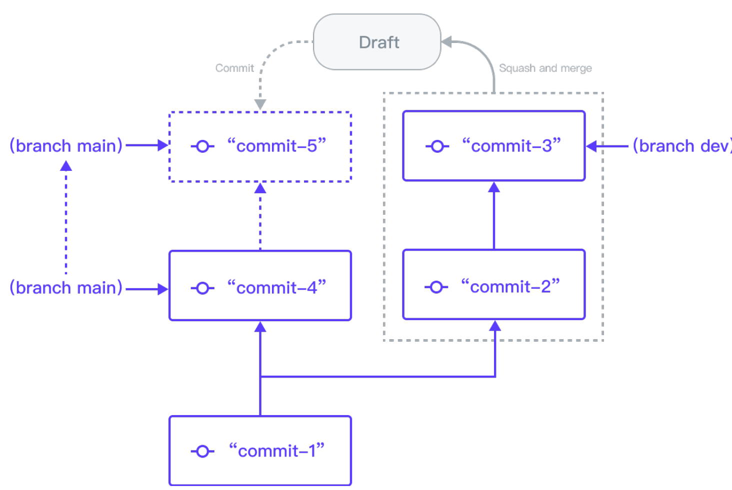

Fig. 1.9: The graphical gas log about the squash and merge operation below.

```python
from tensorbay import GAS
```

(continues on next page)

(continued from previous page)

```
ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
dataset_client = gas.create_dataset("DatasetName")

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

dataset_client.create_branch("dev")
dataset_client.create_draft("draft-2")
dataset_client.commit("commit-2")

dataset_client.create_draft("draft-3")
dataset_client.commit("commit-3")

dataset_client.checkout("main")
dataset_client.create_draft("draft-4")
dataset_client.commit("commit-4")
```

TensorBay SDK allows *squash_and_merge()* by giving the `target_branch_name`:

```
draft_number = dataset_client.squash_and_merge(
    "draft-5",
    description="description",
    source_branch_name="dev",
    target_branch_name="main",
    strategy="override",
)
dataset_client.checkout(draft_number=draft_number)
dataset_client.commit("commit-5")
```

Or `checkout` to the target_branch first. In this case, the current branch is `main`, so we can do squash_and_merge operation directly.

```
draft_number = dataset_client.squash_and_merge(
    "draft-5",
    description="description",
    source_branch_name="dev",
    strategy="override",
)
dataset_client.checkout(draft_number=draft_number)
dataset_client.commit("commit-5")
```

Note: There are three strategies for handling the branch conflict:

1. "abort": abort the opetation;
2. "override": the squashed branch will override the target branch;
3. "skip": keep the origin branch.

1.5 Visualization

Pharos is a plug-in of TensorBay SDK used for local visualization. After finishing the *dataset organization*, users can visualize the organized `Dataset` instance locally using **Pharos**. The visualization result can help users to check whether the dataset is correctly organized.

1.5.1 Install Pharos

To install **Pharos** by **pip**, run the following command:

```
$ pip3 install pharos
```

1.5.2 Pharos Usage

Organize a Dataset

Take the *BSTLD* as an example:

```
from tensorbay.opendataset import BSTLD

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

Visualize the Dataset

```
from pharos import visualize

visualize(dataset)
```

Open the returned URL to see the visualization result.

Note: By default, a pharos server runs locally at `127.0.0.1:5000` and is accessible only from localhost. To change the default setting, the following arguments in `visualize()` can be set:

- `port`: the port the server runs in
- `host`: the host ip to listen on

Visualize the Dataset on Remote Server

Pharos supports accessing the server remotely via a web browser by setting `host` to `"0.0.0.0"`. Then open `http://{external IP}:{port}` in the local browser to get the page. The `external IP` is external ip of the server which pharos runs on.

```
visualize(dataset, host="0.0.0.0", port=5000)
```

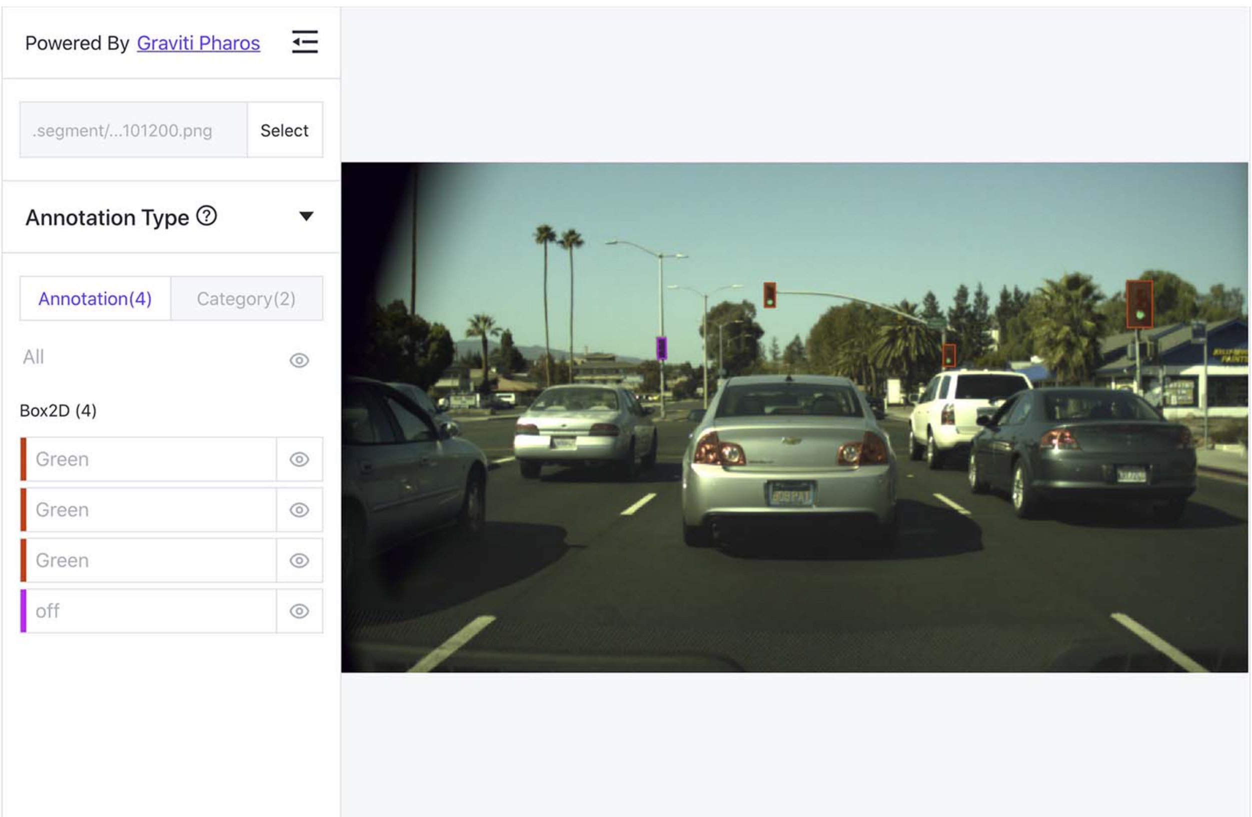

Fig. 1.10: The visualized result of the BSTLD dataset.

1.6 Fusion Dataset

Fusion dataset represents datasets with data collected from multiple sensors. Typical examples of fusion dataset are some autonomous driving datasets, such as nuScenes and KITTI-tracking.

1.6.1 Fusion Dataset Structure

TensorBay also defines a uniform fusion dataset format. This topic explains the related concepts. The TensorBay fusion dataset format looks like:

```
fusion dataset
├── notes
├── catalog
│   ├── subcatalog
│   ├── subcatalog
│   └── ...
├── fusion segment
│   ├── sensors
│   │   ├── sensor
│   │   ├── sensor
│   │   └── ...
│   ├── frame
│   │   ├── data
│   │   └── ...
│   ├── frame
│   │   ├── data
│   │   └── ...
│   └── ...
├── fusion segment
└── ...
```

fusion dataset

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

The corresponding class of fusion dataset is `FusionDataset`.

notes

The notes of the fusion dataset is the same as the notes (*ref*) of the dataset.

catalog & subcatalog in fusion dataset

The catalog of the fusion dataset is the same as the catalog (*ref*) of the dataset.

fusion segment

There may be several parts in a fusion dataset. In TensorBay format, each part of the fusion dataset is stored in one fusion segment. Each fusion segment contains a certain number of frames and multiple sensors, from which the data inside the fusion segment are collected.

The corresponding class of fusion segment is `FusionSegment`.

sensor

Sensor represents the device that collects the data inside the fusion segment. Currently, TensorBay supports four sensor types.(Table. 1.2)

Table 1.2: supported sensors

Supported Sensors	Corresponding Data Type
`Camera`	image
`FisheyeCamera`	image
`Lidar`	point cloud
`Radar`	point cloud

The corresponding class of sensor is `Sensor`.

frame

Frame is the structural level next to the fusion segment. Each frame contains multiple data collected from different sensors at the same time.

The corresponding class of frame is `Frame`.

data in fusion dataset

Each data inside a frame corresponds to a sensor. And the data of the fusion dataset is the same as the data (*ref*) of the dataset.

1.6.2 CADC

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

"CADC" is a fusion dataset with 8 *sensors* including 7 `cameras` and 1 `lidar` , and has *Box3D* type of labels on the point cloud data. (Fig. 1.11). See this page for more details about this dataset.

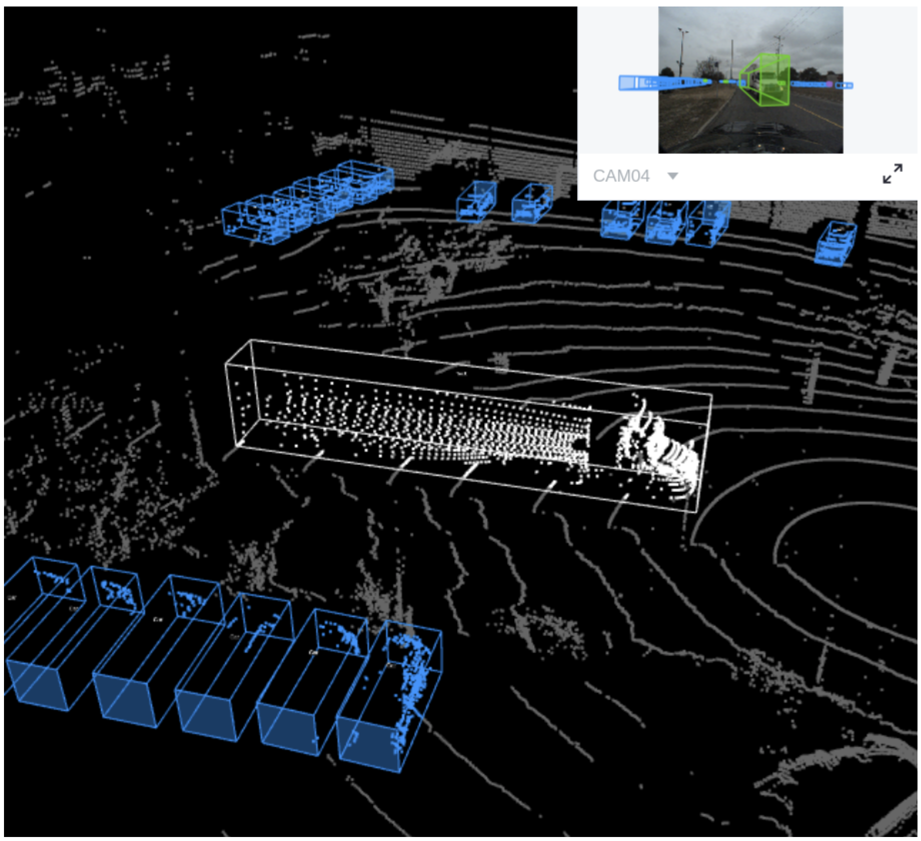

Fig. 1.11: The preview of a point cloud from "CADC" with Box3D labels.

Authorize a Client Instance

First of all, create a GAS client.

```
from tensorbay import GAS
from tensorbay.dataset import FusionDataset

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

Create Fusion Dataset

Then, create a fusion dataset client by passing the fusion dataset name and `is_fusion` argument to the GAS client.

```
gas.create_dataset("CADC", is_fusion=True)
```

List Dataset Names

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

The datasets listed here include both *datasets* and *fusion datasets*.

```
gas.list_dataset_names()
```

Organize Fusion Dataset

Now we describe how to organize the "CADC" fusion dataset by the *FusionDataset* instance before uploading it to TensorBay. It takes the following steps to organize "CADC".

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 "CADC" is *Box3D*, and there are 10 *category* types and 9 *attributes* types.

```
{
    "BOX3D": {
        "isTracking": true,
        "categories": [
            { "name": "Animal" },
            { "name": "Bicycle" },
            { "name": "Bus" },
            { "name": "Car" },
            { "name": "Garbage_Container_on_Wheels" },
            { "name": "Pedestrian" },
            { "name": "Pedestrian_With_Object" },
            { "name": "Traffic_Guidance_Objects" },
            { "name": "Truck" },
            { "name": "Horse and Buggy" }
        ],
```

(continues on next page)

(continued from previous page)

```json
        "attributes": [
            {
                "name": "stationary",
                "type": "boolean"
            },
            {
                "name": "camera_used",
                "enum": [0, 1, 2, 3, 4, 5, 6, 7, null]
            },
            {
                "name": "state",
                "enum": ["Moving", "Parked", "Stopped"],
                "parentCategories": ["Car", "Truck", "Bus", "Bicycle", "Horse_and_Buggy"]
            },
            {
                "name": "truck_type",
                "enum": [
                    "Construction_Truck",
                    "Emergency_Truck",
                    "Garbage_Truck",
                    "Pickup_Truck",
                    "Semi_Truck",
                    "Snowplow_Truck"
                ],
                "parentCategories": ["Truck"]
            },
            {
                "name": "bus_type",
                "enum": ["Coach_Bus", "Transit_Bus", "Standard_School_Bus", "Van_School_Bus"],
                "parentCategories": ["Bus"]
            },
            {
                "name": "age",
                "enum": ["Adult", "Child"],
                "parentCategories": ["Pedestrian", "Pedestrian_With_Object"]
            },
            {
                "name": "traffic_guidance_type",
                "enum": ["Permanent", "Moveable"],
                "parentCategories": ["Traffic_Guidance_Objects"]
            },
            {
                "name": "rider_state",
                "enum": ["With_Rider", "Without_Rider"],
                "parentCategories": ["Bicycle"]
            },
            {
                "name": "points_count",
                "type": "integer",
                "minimum": 0
            }
```

(continues on next page)

(continued from previous page)

```
        ]
    }
}
```

Note: The annotations for "CADC" have tracking information, hence the value of `isTracking` should be set as `True`.

Write the Dataloader

The second step is to write the *dataloader*. The *dataloader* function of "CADC" is to manage all the files and annotations of "CADC" into a `FusionDataset` instance. The *code block* below displays the "CADC" dataloader.

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

"""Dataloader of CADC dataset."""

import json
import os
from datetime import datetime
from typing import Any, Dict, List

import quaternion

from tensorbay.dataset import Data, Frame, FusionDataset
from tensorbay.exception import ModuleImportError
from tensorbay.label import LabeledBox3D
from tensorbay.opendataset._utility import glob
from tensorbay.sensor import Camera, Lidar, Sensors

DATASET_NAME = "CADC"


def CADC(path: str) -> FusionDataset:
    """`CADC <http://cadcd.uwaterloo.ca/index.html>`_ dataset.

    The file structure should be like::

        <path>
            2018_03_06/
                0001/
                    3d_ann.json
                    labeled/
                        image_00/
                            data/
                                0000000000.png
                                0000000001.png
```

(continues on next page)

(continued from previous page)

```
                                ...
                            timestamps.txt
                        ...
                        image_07/
                            data/
                            timestamps.txt
                        lidar_points/
                            data/
                            timestamps.txt
                        novatel/
                            data/
                            dataformat.txt
                            timestamps.txt
                ...
                0018/
                calib/
                    00.yaml
                    01.yaml
                    02.yaml
                    03.yaml
                    04.yaml
                    05.yaml
                    06.yaml
                    07.yaml
                    extrinsics.yaml
                    README.txt
            2018_03_07/
            2019_02_27/

    Arguments:
        path: The root directory of the dataset.

    Returns:
        Loaded `~tensorbay.dataset.dataset.FusionDataset` instance.

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

    dataset = FusionDataset(DATASET_NAME)
    dataset.notes.is_continuous = True
    dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))

    for date in os.listdir(root_path):
        date_path = os.path.join(root_path, date)
        sensors = _load_sensors(os.path.join(date_path, "calib"))
        for index in os.listdir(date_path):
            if index == "calib":
                continue

            segment = dataset.create_segment(f"{date}-{index}")
            segment.sensors = sensors
            segment_path = os.path.join(root_path, date, index)
```

(continues on next page)

(continued from previous page)

```python
            data_path = os.path.join(segment_path, "labeled")

            with open(os.path.join(segment_path, "3d_ann.json"), "r", encoding="utf-8") as fp:
                # The first line of the json file is the json body.
                annotations = json.loads(fp.readline())
            timestamps = _load_timestamps(sensors, data_path)
            for frame_index, annotation in enumerate(annotations):
                segment.append(_load_frame(sensors, data_path, frame_index, annotation, timestamps))

    return dataset


def _load_timestamps(sensors: Sensors, data_path: str) -> Dict[str, List[str]]:
    timestamps = {}
    for sensor_name in sensors.keys():
        data_folder = f"image_{sensor_name[-2:]}" if sensor_name != "LIDAR" else "lidar_points"
        timestamp_file = os.path.join(data_path, data_folder, "timestamps.txt")
        with open(timestamp_file, "r", encoding="utf-8") as fp:
            timestamps[sensor_name] = fp.readlines()

    return timestamps


def _load_frame(
    sensors: Sensors,
    data_path: str,
    frame_index: int,
    annotation: Dict[str, Any],
    timestamps: Dict[str, List[str]],
) -> Frame:
    frame = Frame()
    for sensor_name in sensors.keys():
        # The data file name is a string of length 10 with each digit being a number:
        # 0000000000.jpg
        # 0000000001.bin
        stem = f"{frame_index:010}"

        # Each line of the timestamps file looks like:
        # 2018-03-06 15:02:33.000000000
        timestamp = datetime.strptime(
            timestamps[sensor_name][frame_index][:23], "%Y-%m-%d %H:%M:%S.%f"
        ).timestamp()
        if sensor_name != "LIDAR":
            # The image folder corresponds to different cameras, whose name is likes "CAM00".
            # The image folder looks like "image_00".
            camera_folder = f"image_{sensor_name[-2:]}"
            image_file = f"{stem}.png"

```

(continues on next page)

(continued from previous page)

```
            data = Data(
                os.path.join(data_path, camera_folder, "data", image_file),
                target_remote_path=f"{camera_folder}-{image_file}",
                timestamp=timestamp,
            )
        else:
            data = Data(
                os.path.join(data_path, "lidar_points", "data", f"{stem}.bin"),
                timestamp=timestamp,
            )
            data.label.box3d = _load_labels(annotation["cuboids"])

        frame[sensor_name] = data
    return frame


def _load_labels(boxes: List[Dict[str, Any]]) -> List[LabeledBox3D]:
    labels = []
    for box in boxes:
        dimension = box["dimensions"]
        position = box["position"]

        attributes = box["attributes"]
        attributes["stationary"] = box["stationary"]
        attributes["camera_used"] = box["camera_used"]
        attributes["points_count"] = box["points_count"]

        label = LabeledBox3D(
            size=(
                dimension["y"],  # The "y" dimension is the width from front to back.
                dimension["x"],  # The "x" dimension is the width from left to right.
                dimension["z"],
            ),
            translation=(
                position["x"],  # "x" axis points to the forward facing direction of the␣
↪object.
                position["y"],  # "y" axis points to the left direction of the object.
                position["z"],
            ),
            rotation=quaternion.from_rotation_vector((0, 0, box["yaw"])),
            category=box["label"],
            attributes=attributes,
            instance=box["uuid"],
        )
        labels.append(label)

    return labels


def _load_sensors(calib_path: str) -> Sensors:
    try:
        import yaml  # pylint: disable=import-outside-toplevel
```

(continues on next page)

(continued from previous page)

```
    except ModuleNotFoundError as error:
        raise ModuleImportError(module_name=error.name, package_name="pyyaml") from error

    sensors = Sensors()

    lidar = Lidar("LIDAR")
    lidar.set_extrinsics()
    sensors.add(lidar)

    with open(os.path.join(calib_path, "extrinsics.yaml"), "r", encoding="utf-8") as fp:
        extrinsics = yaml.load(fp, Loader=yaml.FullLoader)

    for camera_calibration_file in glob(os.path.join(calib_path, "[0-9]*.yaml")):
        with open(camera_calibration_file, "r", encoding="utf-8") as fp:
            camera_calibration = yaml.load(fp, Loader=yaml.FullLoader)

        # camera_calibration_file looks like:
        # /path-to-CADC/2018_03_06/calib/00.yaml
        camera_name = f"CAM{os.path.splitext(os.path.basename(camera_calibration_file))[0]}"
        camera = Camera(camera_name)
        camera.description = camera_calibration["camera_name"]

        camera.set_extrinsics(matrix=extrinsics[f"T_LIDAR_{camera_name}"])

        camera_matrix = camera_calibration["camera_matrix"]["data"]
        camera.set_camera_matrix(matrix=[camera_matrix[:3], camera_matrix[3:6], camera_matrix[6:9]])

        distortion = camera_calibration["distortion_coefficients"]["data"]
        camera.set_distortion_coefficients(**dict(zip(("k1", "k2", "p1", "p2", "k3"), distortion)))

        sensors.add(camera)
    return sensors
```

create a fusion dataset

To load a fusion dataset, we first need to create an instance of `FusionDataset`.(L75)

Note that after creating the *fusion dataset*, you need to set the `is_continuous` attribute of `notes` to `True`,(L76) since the *frames* in each *fusion segment* is time-continuous.

load the catalog

Same as dataset, you also need to load the *catalog*.(L77) The catalog file "catalog.json" is in the same directory with dataloader file.

create fusion segments

In this example, we create fusion segments by `dataset.create_segment(SEGMENT_NAME)`.(L86) We manage the data under the subfolder(L33) of the date folder(L32) into a fusion segment and combine two folder names to form a segment name, which is to ensure that frames in each segment are continuous.

add sensors to fusion segments

After constructing the fusion segment, the *sensors* corresponding to different data should be added to the fusion segment.(L87)

In "CADC" , there is a need for projection, so we need not only the name for each sensor, but also the calibration parameters.

And to manage all the `Sensors` (L81, L183) corresponding to different data, the parameters from calibration files are extracted.

`Lidar` sensor only has `extrinsics`, here we regard the lidar as the origin of the point cloud 3D coordinate system, and set the extrinsics as defaults(L189).

To keep the projection relationship between sensors, we set the transform from the camera 3D coordinate system to the lidar 3D coordinate system as `Camera` extrinsics(L205).

Besides `extrinsics()`, `Camera` sensor also has `intrinsics()`, which are used to project 3D points to 2D pixels.

The intrinsics consist of two parts, `CameraMatrix` and `DistortionCoefficients`.(L208-L211)

add frames to segment

After adding the sensors to the fusion segments, the frames should be added into the continuous segment in order(L96).

Each frame contains the data corresponding to each sensor, and each data should be added to the frame under the key of sensor name(L147).

In fusion datasets, it is common that not all data have labels. In "CADC", only point cloud files(Lidar data) have *Box3D* type of labels(L145). See *this page* for more details about Box3D annotation details.

Note: The *CADC dataloader* above uses relative import(L16-L19). 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.

Visualize Dataset

Optionally, the organized dataset can be visualized by **Pharos**, which is a TensorBay SDK plug-in. This step can help users to check whether the dataset is correctly organized. Please see *Visualization* for more details.

Upload Fusion Dataset

After you finish the *dataloader* and organize the "CADC" into a `FusionDataset` instance, you can upload it to TensorBay for sharing, reuse, etc.

```
# fusion_dataset is the one you initialized in "Organize Fusion Dataset" section
fusion_dataset_client = gas.upload_dataset(fusion_dataset, jobs=8)
fusion_dataset_client.commit("initial commit")
```

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 Fusion Dataset

Now you can read "CADC" dataset from TensorBay.

```
fusion_dataset = FusionDataset("CADC", gas)
```

In *dataset* "CADC", there are lots of *FusionSegments*: `2018_03_06/0001`, `2018_03_07/0001`, …

You can get the segment names by list them all.

```
fusion_dataset.keys()
```

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

```
fusion_segment = fusion_dataset["2018_03_06/0001"]
```

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

In the `2018_03_06/0001` *fusion segment*, there are several *sensors*. You can get all the sensors by accessing the `sensors` of the `FusionSegment`.

```
sensors = fusion_segment.sensors
```

In each *fusion segment*, there are a sequence of *frames*. You can get one by index.

```
frame = fusion_segment[0]
```

In each *frame*, there are several *data* corresponding to different sensors. You can get each data by the corresponding sensor name.

```
for sensor_name in sensors.keys():
    data = frame[sensor_name]
```

In "CADC", only *data* under `Lidar` has a sequence of *Box3D* annotations. You can get one by index.

```
lidar_data = frame["LIDAR"]
label_box3d = lidar_data.label.box3d[0]
category = label_box3d.category
attributes = label_box3d.attributes
```

There is only one label type in "CADC" 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 some of the attributes in "attributes" list of *catalog.json*.

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

Delete Fusion Dataset

To delete "CADC", run the following code:

```
gas.delete_dataset("CADC")
```

1.7 Storage Config

TensorBay supports two storage config modes:

- GRAVITI Storage Config: storage config provided by graviti.
- Authorized Storage Config: storage config provided by userself.

1.7.1 GRAVITI Storage Config

In graviti storage mode, the data is stored in graviti storage space on TensorBay.

1.7.2 Authorized Storage Config

When using authorized storage config, datasets are stored on user's storage space and are only indexed to the TensorBay. See authorized storage instruction for details about how to configure authorized storage on TensorBay.

TensorBay supports both authorize cloud storage and authorize local storage.

Authorized Cloud Storage

TensorBay SDK supports following methods to configure authorized cloud storage.

- *`create_oss_storage_config()`*
- *`create_s3_storage_config()`*
- *`create_azure_storage_config()`*

For example:

```
gas.create_oss_storage_config(
    "oss_config",
    "tests",
    endpoint="<YOUR_ENDPOINT>",  # like oss-cn-qingdao.aliyuncs.com
    accesskey_id="<YOUR_ACCESSKEYID>",
    accesskey_secret="<YOUR_ACCESSKEYSECRET>",
    bucket_name="<YOUR_BUCKETNAME>",
)
```

TensorBay SDK supports a method to list a user's all previous configurations.

```
gas.list_auth_storage_configs()
```

Create Authorized Storage Dataset

Create a dataset with authorized cloud storage:

```
dataset_client = gas.create_dataset("dataset_name", config_name="config_name")
```

Import Cloud Files into Authorized Storage Dataset

Take the following original cloud storage directory as an example:

```
data/
├── images/
│   ├── 00001.png
│   ├── 00002.png
│   └── ...
├── labels/
│   ├── 00001.json
│   ├── 00002.json
│   └── ...
└── ...
```

Get a cloud client.

```
from tensorbay import GAS

gas = GAS("Accesskey-*****")
cloud_client = gas.get_cloud_client("config_name")
```

Import the AuthData from original cloud storage and load label file to an authorized storage dataset.

```
import json

from tensorbay.dataset import Dataset
from tensorbay.label import Classification

# Use AuthData to organize a dataset by the "Dataset" class before importing.
dataset = Dataset("DatasetName")

```

(continues on next page)

(continued from previous page)

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

images = cloud_client.list_auth_data("data/images/")
labels = cloud_client.list_auth_data("data/labels/")

for auth_data, label in zip(images, labels):
    with label.open() as fp:
        auth_data.label.classification = Classification.loads(json.load(fp))
    segment.append(auth_data)

dataset_client = gas.upload_dataset(dataset, jobs=8)
```

Important: Files will be copied from original directory to the authorized storage dataset path, thus the storage space will be doubled.

Note: Set the authorized storage dataset path the same as original cloud storage directory could speed up the import action. For example, set the config path of above dataset to `data/images`.

Authorized Local Storage

If you want to use TensorBay service and have the data stored locally at the same time, TensorBay supports authorized local storage config.

Before creating the local storage config via *create_local_storage_config()*, you need to start a local storage service. Please contact us on TensorBay for more information.

```
gas.create_local_storage_config(
    name="local_storage_config",
    file_path="<path to store the datasets>",
    endpoint="<external IP address of the local storage service>",
)
```

Then create an authorized local storage dataset with the config.

```
dataset_client = gas.create_dataset("dataset_name", config_name="local_storage_config")
```

Other operations such as uploading data and reading data, are the same as datasets created by default, except that the uploaded data is stored under the local storage.

1.8 Request Configuration

This topic introduces the currently supported `Config` options(Table. 1.3) for customizing request. Note that the default settings can satisfy most use cases.

Table 1.3: Requests Configuration Tables

Variables	Description
max_retries	The number of maximum retry times of the request. If the request method is one of the allowed_retry_methods and the response status is one of the allowed_retry_status, then the request can auto-retry *max_retries* times. Scenario: Enlarge it when under poor network quality. Default: 3 times.
allowed_retry_methods	The allowed methods for retrying request. Default: [“HEAD”, “OPTIONS”, “POST”, “PUT”]
allowed_retry_status	The allowed status for retrying request. Default: [429, 500, 502, 503, 504]
timeout	The number of seconds before the request times out. Scenario: Enlarge it when under poor network quality. Default: 30 seconds.
is_internal	Whether the request is from internal or not. Scenario: Set it to True for quicker network speed when datasets and cloud servers are in the same region. See *Use Internal Endpoint* for details. Default: False

1.8.1 Usage

```
from tensorbay import GAS
from tensorbay.client import config

# Enlarge timeout and max_retries of configuration.
config.timeout = 40
config.max_retries = 4

gas = GAS("<YOUR_ACCESSKEY>")

# The configs will apply to all the requests sent by TensorBay SDK.
gas.list_dataset_names()
```

1.9 Use Internal Endpoint

This topic describes how to use the internal endpoint when using TensorBay.

1.9.1 Region and Endpoint

For a cloud storage service platform, a region is a collection of its resources in a geographic area. Each region is isolated and independent of the other regions. Endpoints are the domain names that other services can use to access the cloud platform. Thus, there are mappings between regions and endpoints. Take OSS as an example, the endpoint for region **China (Hangzhou)** is *oss-cn-hangzhou.aliyuncs.com*.

Actually, the endpoint mentioned above is the public endpoint. There is another kind of endpoint called the internal endpoint. The internal endpoint can be used by other cloud services in the **same region** to access cloud storage services. For example, the internal endpoint for region **China (Hangzhou)** is *oss-cn-hangzhou-internal.aliyuncs.com*.

Much quicker internet speed is the most important benefit of using an internal endpoint. Currently, TensorBay supports using the internal endpoint of OSS for operations such as uploading and reading datasets.

1.9.2 Usage

If the endpoint of the cloud server is the same as the TensorBay storage, set *is_internal* to *True* to use the internal endpoint for obtaining a faster network speed.

```
from tensorbay import GAS
from tensorbay.client import config
from tensorbay.dataset import Data, Dataset

# Set is_internal to True for using internal endpoint.
config.is_internal = True

gas = GAS("<YOUR_ACCESSKEY>")

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

segment = dataset.create_segment()
```

(continues on next page)

(continued from previous page)

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

# All the data will be uploaded through internal endpoint.
dataset_client = gas.upload_dataset(dataset, jobs=8)

dataset_client.commit("Initial commit")
```

1.10 Profilers

This topic describes how to use *Profile* to record speed statistics.

1.10.1 Usage

You can save the statistical record to a txt, csv or json file.

```
from tensorbay.client import profile

# Start record.
with profile as pf:
    # <Your Program>

    # Save the statistical record to a file.
    pf.save("summary.txt", file_type="txt")
```

Set `multiprocess=True` to record the multiprocessing program.

```
# Start record.
profile.start(multiprocess=True)

# <Your Program>

# Save the statistical record to a file.
profile.save("summary.txt", file_type="txt")
profile.stop()
```

The above action would save a summary.txt file and the result is as follows:

```
|Path                    |totalTime (s) |callNumber   |avgTime (s) |totalResponseLength ␣
↪|totalFileSize (B)|
|[GET] data06/labels     |11.239        |25           |0.450       |453482              ␣
↪|0                |
|[GET] data06/data/urls  |16.739        |25           |0.670       |794545              ␣
↪|0                |
|[POST] oss-cn-shanghai  |0.567         |10           |0.057       |0                   ␣
↪|8058707          |
```

Note: The *profile* will only record statistics of the interface that interacts with Tensorbay.

1.11 Cache

This topic describes how to use cache while opening remote data on Tensorbay.

While using online data, sometimes it may be neccessary to use the entire dataset multiple times, such as training model.

This would cause redundant requests and responses between the local computer and TensorBay, and cost extra time.

Therefore, TensorBaySDK provides caching to speed up data access and reduce repeated requests.

1.11.1 Get Remote Dataset

To use the cache, first get the remote dataset on TensorBay.

```
from tensorbay import GAS
from tensorbay.dataset import Dataset

ACCESS_KEY = "Accesskey-*****"
gas = GAS(ACCESS_KEY)
dataset = Dataset("<DatasetName>", gas)
```

1.11.2 Enable Cache

Then use *enable_cache()* to start using cache for this dataset. The cache path is set in the temporary directory by default, which differs according to the system.

```
dataset.enable_cache()
```

It's also feasible to pass a custom cache path to the function as below.

```
dataset.enable_cache("path/to/cache/folder")
```

Note: Please make sure there is enough free storage space to cache the dataset.

Use *cache_enabled* to check whether the cache is in use.

```
print(dataset.cache_enabled)
# True
```

Note: Cache is not available for datasets in draft status. The `dataset.cache_enabled` will remain `False` for datasets in draft status, even if the cache has already been set by `dataset.enable_cache()`.

1.11.3 Use Data

After enabling the cache, use the data as desired. Note that the cache works when the `data.open()` method is called, and only data and mask labels will be cached.

```
segment = dataset[0]
MAX_EPOCH = 100
for epoch in range(MAX_EPOCH):
    for data in segment:
        data.open()
        # code using opened data here
```

1.11.4 Delete Cache Data

After use, according to the cache path, the cache data can be deleted as needed.

Note that if the default cache path is used, the cache will be removed automatically when the computer restarts.

1.12 PaddlePaddle

This topic describes how to integrate TensorBay dataset with PaddlePaddle Pipeline using the MNIST Dataset as an example.

The typical method to integrate TensorBay dataset with PaddlePaddle is to build a "Segment" class derived from `paddle.io.Dataset`.

```
from paddle.io import DataLoader, Dataset
from paddle.vision import transforms
from PIL import Image

from tensorbay import GAS
from tensorbay.dataset import Dataset as TensorBayDataset


class MNISTSegment(Dataset):
    """class for wrapping a MNIST segment."""

    def __init__(self, gas, segment_name, transform):
        super().__init__()
        self.dataset = TensorBayDataset("MNIST", gas)
        self.segment = self.dataset[segment_name]
        self.category_to_index = self.dataset.catalog.classification.get_category_to_
↪index()
        self.transform = transform

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

    def __getitem__(self, idx):
        data = self.segment[idx]
        with data.open() as fp:
```

(continues on next page)

(continued from previous page)

```
            image_tensor = self.transform(Image.open(fp))

        return image_tensor, self.category_to_index[data.label.classification.category]
```

Using the following code to create a PaddlePaddle dataloader and run it:

```
ACCESS_KEY = "Accesskey-*****"

to_tensor = transforms.ToTensor()
normalization = transforms.Normalize(mean=[0.485], std=[0.229])
my_transforms = transforms.Compose([to_tensor, normalization])

train_segment = MNISTSegment(GAS(ACCESS_KEY), segment_name="train", transform=my_
↪transforms)
train_dataloader = DataLoader(train_segment, batch_size=4, shuffle=True, num_workers=0)

for index, (image, label) in enumerate(train_dataloader):
    print(f"{index}: {label}")
```

1.13 PyTorch

This topic describes how to integrate TensorBay dataset with PyTorch Pipeline using the MNIST Dataset as an example.

The typical method to integrate TensorBay dataset with PyTorch is to build a "Segment" class derived from `torch.utils.data.Dataset`.

```
from PIL import Image
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms

from tensorbay import GAS
from tensorbay.dataset import Dataset as TensorBayDataset


class MNISTSegment(Dataset):
    """class for wrapping a MNIST segment."""

    def __init__(self, gas, segment_name, transform):
        super().__init__()
        self.dataset = TensorBayDataset("MNIST", gas)
        self.segment = self.dataset[segment_name]
        self.category_to_index = self.dataset.catalog.classification.get_category_to_
↪index()
        self.transform = transform

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

    def __getitem__(self, idx):
```

(continues on next page)

(continued from previous page)

```
        data = self.segment[idx]
        with data.open() as fp:
            image_tensor = self.transform(Image.open(fp))

        return image_tensor, self.category_to_index[data.label.classification.category]
```

Using the following code to create a PyTorch dataloader and run it:

```
ACCESS_KEY = "Accesskey-*****"

to_tensor = transforms.ToTensor()
normalization = transforms.Normalize(mean=[0.485], std=[0.229])
my_transforms = transforms.Compose([to_tensor, normalization])

train_segment = MNISTSegment(GAS(ACCESS_KEY), segment_name="train", transform=my_
↪transforms)
train_dataloader = DataLoader(train_segment, batch_size=4, shuffle=True, num_workers=4)

for index, (image, label) in enumerate(train_dataloader):
    print(f"{index}: {label}")
```

1.14 TensorFlow

This topic describes how to integrate TensorBay dataset with TensorFlow Pipeline using the MNIST Dataset as an example.

The typical method to integrate TensorBay dataset with TensorFlow is to build a callable "Segment" class.

```
import numpy as np
import tensorflow as tf
from PIL import Image
from tensorflow.data import Dataset

from tensorbay import GAS
from tensorbay.dataset import Dataset as TensorBayDataset


class MNISTSegment:
    """class for wrapping a MNIST segment."""

    def __init__(self, gas, segment_name):
        self.dataset = TensorBayDataset("MNIST", gas)
        self.segment = self.dataset[segment_name]
        self.category_to_index = self.dataset.catalog.classification.get_category_to_
↪index()

    def __call__(self):
        """Yield an image and its corresponding label.

        Yields:
            image_tensor: the tensorflow sensor of the image.
```

(continues on next page)

(continued from previous page)

```
            category_tensor: the tensorflow sensor of the category.

        """
        for data in self.segment:
            with data.open() as fp:
                image_tensor = tf.convert_to_tensor(
                    np.array(Image.open(fp)) / 255, dtype=tf.float32
                )
            category = self.category_to_index[data.label.classification.category]
            category_tensor = tf.convert_to_tensor(category, dtype=tf.int32)
            yield image_tensor, category_tensor
```

Using the following code to create a TensorFlow dataset and run it:

```
ACCESS_KEY = "Accesskey-*****"

dataset = Dataset.from_generator(
    MNISTSegment(GAS(ACCESS_KEY), "train"),
    output_signature=(
        tf.TensorSpec(shape=(28, 28), dtype=tf.float32),
        tf.TensorSpec(shape=(), dtype=tf.int32),
    ),
).batch(4)

for index, (image, label) in enumerate(dataset):
    print(f"{index}: {label}")
```

1.15 Getting Started with CLI

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

TensorBay CLI supports:

- list, create and delete operations for dataset, segment and data.
- uploading data to TensorBay.
- version control operations with branch, tag, draft and commit.
- showing commit logs of dataset on TensorBay.

1.15.1 Installation

To use TensorBay CLI, please install TensorBay SDK first.

```
$ pip3 install tensorbay
```

1.15.2 Authentication

An accessKey is used for identification when using TensorBay to operate datasets.

Set the accessKey into configuration:

```
$ gas auth [ACCESSKEY]
```

To show authentication information:

```
$ gas auth --get
```

1.15.3 TBRN

TensorBay Resource Name(TBRN) uniquely defines the resource stored in TensorBay. TBRN begins with `tb:`. See more details in *TBRN*. The following is the general format for TBRN:

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

1.15.4 Usage

CLI: Create a Dataset

```
$ gas dataset tb:<dataset_name>
```

CLI: List Dataset Names

```
$ gas dataset
```

CLI: Create a Draft

```
$ gas draft tb:<dataset_name> [-m <title>]
```

CLI: List Drafts

```
$ gas draft -l tb:<dataset_name>
```

CLI: Upload a File To the Dataset

```
$ gas cp <local_path> tb:<dataset_name>#<draft_number>:<segment_name>
```

CLI: Commit the Draft

```
$ gas commit tb:<dataset_name>#<draft_number> [-m <title>]
```

1.15.5 Profile

For users with multiple TensorBay accounts or different workspaces, CLI provides profiles to easily authenticate and use different accessKeys.

Set the accessKey into the specific profile, and show the specific profile's authentication information:

```
$ gas -p <profile_name> auth [ACCESSKEY]
$ gas -p <profile_name> auth -g
```

After authentication, the profiles can be used to execute other commands:

```
$ gas -p <profile_name> <command>
```

For example, list all the datasets with the given profile's accessKey:

```
$ gas -p <profile_name> ls
```

For users who want to use a temporary accessKey, CLI provides `-k` option to override the authentication:

```
$ gas -k <Accesskey> <command>
```

For example, list all the datasets with the given accessKey:

```
$ gas -k <AccessKey> ls
```

1.16 TensorBay Resource Name

TensorBay Resource Name(TBRN) uniquely identifies the resource stored in TensorBay. All TBRN begins with `tb:`.

1. Identify a dataset

```
tb:<dataset_name>
```

For example, the following TBRN means the dataset "VOC2012".

```
tb:VOC2012
```

2. Identify a segment

```
tb:<dataset_name>:<segment_name>
```

For example, the following TBRN means the "train" segment of dataset "VOC2012".

```
tb:VOC2010:train
```

3. Identify a file

```
tb:<dataset_name>:<segment_name>://<remote_path>
```

For example, the following TBRN means the file “2012_004330.jpg” under “train” segment in dataset “VOC2012”.

```
tb:VOC2012:train://2012_004330.jpg
```

1.16.1 TBRN With Version Info

The version information can also be included in the TBRN when using *version control* feature.

1. Include revision info:

A TBRN can include revision info in the following format:

```
tb:<dataset_name>@<revision>[:<segment_name>][://<remote_path>]
```

For example, the following TBRN means the main branch of dataset “VOC2012”.

```
tb:VOC2010@main
```

2. Include draft info:

A TBRN can include draft info in the following format:

```
tb:<dataset_name>#<draft_number>[:<segment_name>][://<remote_path>]
```

For example, the following TBRN means the 1st draft of dataset “VOC2012”.

```
tb:VOC2012#1
```

Note that if neither revision nor draft number is given, a TBRN will refer to the default branch.

1.17 CLI Commands

The following table lists the currently supported CLI commands.(Table. 1.4).

Table 1.4: CLI Commands

Commands	Description
gas auth	authentication operations.
gas config	config operations
gas dataset	dataset operations.
gas ls	list operations.
gas cp	copy operations.
gas rm	remove operations.
gas draft	draft operations.
gas commit	commit operations.
gas tag	tag operations.
gas log	log operations.
gas branch	branch operations

1.17.1 gas auth

Work with authentication operations.

Authenticate the accesskey of the TensorBay account. If the accesskey is not provided, interactive authentication will be launched.

```
$ gas auth [ACCESSKEY]
```

Get the authentication information.

```
$ gas auth --get [--all]
```

Unset the authentication information.

```
$ gas auth --unset [--all]
```

1.17.2 gas config

Work with configuration operations.

`gas config` supports modifying the configurations about network request and editor.

Add a single configuration, see the available keys and corresponding values about network request at *request_configuration*.

```
$ gas config [key] [value]
```

For example:

```
$ gas config editor vim
$ gas config max_retries 5
```

Show all the configurations.

```
$ gas config
```

Show a single configuration.

```
$ gas config [key]
```

For example:

```
$ gas config editor
```

Unset a single configuration.

```
$ gas config --unset <key>
```

For example:

```
$ gas config --unset editor
```

1.17.3 gas dataset

Work with dataset operations.

Create a dataset.

```
$ gas dataset tb:<dataset_name>
```

List all datasets.

```
$ gas dataset
```

Delete a dataset.

```
$ gas dataset -d tb:<dataset_name>
```

1.17.4 gas ls

Work with list operations.

List the segments of a dataset.(default branch)

```
$ gas ls tb:<dataset_name>
```

List the segments of a specific dataset *revision*.

```
$ gas ls tb:<dataset_name>@<revision>
```

List the segments of a specific dataset draft.

See *gas draft* for more information.

```
$ gas ls tb:<dataset_name>#<draft_number>
```

List all files of a segment.

```
$ gas ls tb:<dataset_name>:<segment_name>
$ gas ls tb:<dataset_name>@<revision>:<segment_name>
$ gas ls tb:<dataset_name>#<draft_number>:<segment_name>
```

Get a certain file.

```
$ gas ls tb:<dataset_name>:<segment_name>://<remote_path>
$ gas ls tb:<dataset_name>@<revision>:<segment_name>://<remote_path>
$ gas ls tb:<dataset_name>#<draft_number>:<segment_name>://<remote_path>
```

1.17.5 gas cp

Work with copy operations.

Upload a file to a segment. The `local_path` refers to a file.

The target dataset must be in draft status, see *gas draft* for more information.

```
$ gas cp <local_path> tb:<dataset_name>#<draft_number>:<segment_name>
```

Upload files to a segment. The `local_path` refers to a directory.

```
$ gas cp -r <local_path> tb:<dataset_name>#<draft_number>:<segment_name>
```

Upload a file to a segment with a given `remote_path`, which is the target path on TensorBay. The `local_path` can refer to only one file.

```
$ gas cp <local_path> tb:<dataset_name>#<draft_number>:<segment_name>://<remote_path>
```

1.17.6 gas rm

Work with remove operations.

Remove a segment.

The target dataset must be in draft status, see *gas draft* for more information.

```
$ gas rm -r tb:<dataset_name>#<draft_number>:<segment_name>
```

Remove a file.

```
$ gas rm tb:<dataset_name>#<draft_number>:<segment_name>://<remote_path>
```

1.17.7 gas draft

Work with *draft* operations.

Create a draft with a title.

```
$ gas draft tb:<dataset_name> [-m <title>]
```

List the drafts of a dataset.

```
$ gas draft -l tb:<dataset_name>
```

Edit the draft of a dataset.

```
$ gas draft -e tb:<dataset_name>#<draft_number> [-m <title>]
```

Close the draft of a dataset.

```
$ gas draft -c tb:<dataset_name>#<draft_number>
```

1.17.8 gas commit

Work with commit operations.

Commit a *draft* with a title.

```
$ gas commit tb:<dataset_name>#<draft_number> [-m <title>]
```

1.17.9 gas tag

Work with *tag* operations.

Create a tag on the current commit or a specific *revision*.

```
$ gas tag tb:<dataset_name> <tag_name>
$ gas tag tb:<dataset_name>@<revision> <tag_name>
```

List all tags.

```
$ gas tag tb:<dataset_name>
```

Delete a tag.

```
$ gas tag -d tb:<dataset_name>@<tag_name>
```

1.17.10 gas log

Work with log operations.

Show the commit logs.

```
$ gas log tb:<dataset_name>
```

Show commit logs from a certain *revision*.

```
$ gas log tb:<dataset_name>@<revision>
```

Limit the number of commit logs to show.

```
$ gas log -n <number> tb:<dataset_name>
$ gas log --max-count <number> tb:<dataset_name>
```

Show commit logs in oneline format.

```
$ gas log --oneline tb:<dataset_name>
```

Show commit logs of all revisions.

```
$ gas log --all tb:<dataset_name>
```

Show graphical commit logs.

```
$ gas log --graph tb:<dataset_name>
```

1.17.11 gas branch

Work with *branch* operations.

Create a new branch from the default branch.

```
$ gas branch tb:<dataset_name> <branch_name>
```

Create a new branch from a certain *revision*.

```
$ gas branch tb:<dataset_name>@<revision> <branch_name>
```

Show all branches.

```
$ gas branch tb:<dataset_name>
```

Delete a branch.

```
$ gas branch --delete tb:<dataset_name>@<branch_name>
```

1.18 Shell Completion

The completion of CLI is supported by the completion of `click`, see details in v7.x and v8.x click documentations.

CLI provides tab completion support for Bash (version not lower than 4.4), Zsh, and Fish. It is possible to add support for other shells too.

Shell completion suggests command names and option names. Options are only listed if at least a dash has been entered.

Here is an example of completion:

```
$ gas <TAB><TAB>
auth     -- Authenticate the accessKey of gas.
branch   -- List, create or delete branches.
commit   -- Commit drafts.
config   -- Configure the options when using gas CLI.
cp       -- Copy local data to a remote path.
dataset  -- List, create or delete datasets.
draft    -- List or create drafts.
log      -- Show commit logs.
ls       -- List data under the path.
rm       -- Remove the remote data.
```

(continues on next page)

(continued from previous page)

```
tag      -- List, create or delete tags.
$ gas auth -<TAB><TAB>
--get    -g  -- Get the accesskey of the profile
--status -s  -- Get the user info and accesskey of the profile
--unset  -u  -- Unset the accesskey of the profile
--all    -a  -- All the auth info
--help       -- Show this message and exit.
```

Note: The result may differ with different versions of `click` or shell.

1.18.1 Activation

Completion is only available if `tensorbay` is installed and invoked through `gas`, not through the `python` command.

In order for completion to be used, the user needs to register a special function with their shell. The script is different for every shell. The built-in shells are `bash`, `zsh`, and `fish`. The following instructions will lead user to configure the completion:

Before configuring completion, the user needs to check the version of `click`:

```
$ pip show click
```

Activation for Click 7.x

For Bash: Add this to `~/.bashrc`:

```
eval "$(_GAS_COMPLETE=source_bash gas)"
```

For Zsh: Add this to `~/.zshrc`:

```
eval "$(_GAS_COMPLETE=source_zsh gas)"
```

For Fish: Add this to `~/.config/fish/completions/gas.fish`:

```
eval (env _GAS_COMPLETE=source_fish gas)
```

Activation for Click 8.x

For Bash: Add this to `~/.bashrc`:

```
eval "$(_GAS_COMPLETE=bash_source gas)"
```

For Zsh: Add this to `~/.zshrc`:

```
eval "$(_GAS_COMPLETE=zsh_source gas)"
```

For Fish: Add this to `~/.config/fish/completions/gas.fish`:

```
eval (env _GAS_COMPLETE=fish_source gas)
```

1.18.2 Activation Script

Using `eval` means that the command is invoked and evaluated every time a shell is started, which can delay shell responsiveness. Using activation script is faster than using `eval`: write the generated script to a file, then source that.

Activation Script for Click 7.x

For Bash: Save the script somewhere.

```
_GAS_COMPLETE=source_bash gas > ~/.gas-complete.bash
```

Source the file in `~/.bashrc`.

```
. ~/.gas-complete.bash
```

For Zsh: Save the script somewhere.

```
_GAS_COMPLETE=source_zsh gas > ~/.gas-complete.zsh
```

Source the file in `~/.zshrc`.

```
. ~/.gas-complete.zsh
```

For Fish: Add the file to the completions directory:

```
_GAS_COMPLETE=source_fish gas > ~/.config/fish/completions/gas-complete.fish
```

Activation Script for Click 8.x

For Bash: Save the script somewhere.

```
_GAS_COMPLETE=bash_source gas > ~/.gas-complete.bash
```

Source the file in `~/.bashrc`.

```
. ~/.gas-complete.bash
```

For Zsh: Save the script somewhere.

```
_GAS_COMPLETE=zsh_source gas > ~/.gas-complete.zsh
```

Source the file in `~/.zshrc`.

```
. ~/.gas-complete.zsh
```

For Fish: Save the script to `~/.config/fish/completions/gas.fish`:

```
_GAS_COMPLETE=fish_source gas > ~/.config/fish/completions/gas.fish
```

Note: After modifying the shell config, the user needs to start a new shell or source the modified files in order for the changes to be loaded.

1.19 Sextant

TensorBay SDK supports methods to interact with sextant application. See authorized storage instruction for details about how to start.

1.19.1 Authorize a Sextant Instance

```
from tensorbay.apps.sextant import Sextant

sextant = Sextant("<YOUR_ACCESSKEY>")
```

1.19.2 List or get benchmark

```
# list all benchmarks.
benchmarks = sextant.list_benchmarks()

# get benchmark with given name.
benchmark = sextant.get_benchmark("test_01")
```

1.19.3 Create a evaluation

A evaluation must be created by one commit.

```
from tensorbay import GAS

gas = GAS("<YOUR_ACCESSKEY>")
dataset_client = gas.get_dataset("dataset_name")
dataset_client.checkout(revision="branch/tag/commitId")
evaluation = benchmark.create_evaluation(dataset_client.dataset_id, dataset_client.
↪status.commit_id)
```

1.19.4 List all evaluations

```
evaluations = benchmark.list_evaluations()
evaluation = evaluations[0]
```

1.19.5 get evaluation result

```
result = evaluation.get_result()
```

The details of the result structure for the evaluation are as follows:

```
{
    "result": {
        "categories": {
```

(continues on next page)

(continued from previous page)

```
            "aeroplane": {
                "AP": 0.3,
                "averageIoU": 0.71,
                "pr": {
                    "precision": [0.87, 0.8, ...],
                    "recall": [0.001, 0.0045, ...]
                }
            },
            ...
        },
        "overall": {
            "averageIoU": 0.7,
            "mAP": 0.2,
            "pr": {
                "precision": [0.95, 0.91, ...],
                "recall": [0.0036, 0.01, ...]
            }
        }
    }
}
```

Note: Benchmark can only be created with tensorbay website now.

1.20 Glossary

1.20.1 accesskey

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

To obtain an accesskey, 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*.

1.20.2 basehead

The basehead is the string for recording the two relative versions(commits or drafts) in the format of “base…head”.

The basehead param is comprised of two parts: base and head. Both must be *revision* or draft number in dataset. The terms “head” and “base” are used as they normally are in Git.

The head is the version which changes are on. The base is the version of which these changes are based.

1.20.3 branch

Similar to git, a branch is a lightweight pointer to one of the commits.

Every time a *commit* is submitted, the main branch pointer moves forward automatically to the latest commit.

1.20.4 commit

Similar with Git, a commit is a version of a dataset, which contains the changes compared with the former commit.

Each commit has a unique commit ID, which is a uuid in a 36-byte hexadecimal string. A certain commit of a dataset can be accessed by passing the corresponding commit ID or other forms of *revision*.

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.

1.20.5 continuity

Continuity is a characteristic to describe the data within a *dataset* or a *fusion dataset*.

A dataset is continuous means the data in each segment of the dataset is collected over a continuous period of time and the collection order is indicated by the data paths or frame indexes.

The continuity can be set in *notes*.

Only continuous datasets can have *tracking* labels.

1.20.6 dataloader

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

Here are some dataloader examples of datasets with different label types and continuity(Table. 1.5).

Table 1.5: Dataloaders

Dataloaders	Description
LISA Traffic Light Dataloader	This example is the dataloader of LISA Traffic Light Dataset, which is a continuous dataset with *Box2D* label.
Dogs vs Cats Dataloader	This example is the dataloader of Dogs vs Cats Dataset, which is a dataset with *Classification* label.
BSTLD Dataloader	This example is the dataloader of BSTLD Dataset, which is a dataset with *Box2D* label.
Neolix OD Dataloader	This example is the dataloader of Neolix OD Dataset, which is a dataset with *Box3D* label.
Leeds Sports Pose Daraloader	This example is the dataloader of Leeds Sports Pose Dataset, which is a dataset with *Keypoints2D* label.

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*.

1.20.7 dataset

A uniform dataset format defined by TensorBay, which only contains one type of data collected from one sensor or without sensor information. According to the time continuity of data inside the dataset, a dataset can be a discontinuous dataset or a continuous dataset. *Notes* can be used to specify whether a dataset is continuous.

The corresponding class of dataset is `Dataset`.

See *Dataset Structure* for more details.

1.20.8 diff

TensorBay supports showing the status difference of the relative resource between commits or drafts in the form of diff.

1.20.9 draft

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

A draft is created based on a *branch*, 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, create a draft, edit the dataset and commit the draft.

1.20.10 fusion dataset

A uniform dataset format defined by Tensorbay, which contains data collected from multiple sensors.

According to the time continuity of data inside the dataset, a fusion dataset can be a discontinuous fusion dataset or a continuous fusion dataset. *Notes* can be used to specify whether a fusion dataset is continuous.

The corresponding class of fusion dataset is `FusionDataset`.

See *Fusion Dataset Structure* for more details.

1.20.11 revision

Similar to Git, a revision is a reference to a single *commit*. And many methods in TensorBay SDK take revision as an argument.

Currently, a revision can be in the following forms:

1. A full *commit* ID.
2. A *tag*.
3. A *branch*.

1.20.12 tag

TensorBay SDK has the ability to tag the specific *commit* in a dataset's history as being important. Typically, people use this functionality to mark release points (v1.0, v2.0 and so on).

1.20.13 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 there is an image `000000.jpg` under the `train` segment of a dataset named `example`, then the TBRN of this image should be:

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

1.20.14 tracking

Tracking is a characteristic to describe the labels within a *dataset* or a *fusion dataset*.

The labels of a dataset are tracking means the labels contain tracking information, such as tracking ID, which is used for tracking tasks.

Tracking characteristic is stored in *catalog*, please see *Label Format* for more details.

1.21 Dataset Structure

For ease of use, TensorBay defines a uniform dataset format. This topic explains the related concepts. The TensorBay dataset format looks like:

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

1.21.1 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`.

1.21.2 notes

Notes contains the basic information of a dataset, including

- the time continuity of the data inside the dataset
- the fields of bin point cloud files inside the dataset

The corresponding class of notes is `Notes`.

1.21.3 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 Subcatalog only stores label meta information of one label type, including whether the corresponding annotation has tracking information.

Here are some catalog examples of datasets with different label types and a dataset with tracking annotations(Table. 1.6).

Table 1.6: Catalogs

Catalogs	Description
elpv Catalog	This example is the catalog of elpv Dataset, which is a dataset with *Classification* label.
BSTLD Catalog	This example is the catalog of BSTLD Dataset, which is a dataset with *Box2D* label.
Neolix OD Catalog	This example is the catalog of Neolix OD Dataset, which is a dataset with *Box3D* label.
Leeds Sports Pose Catalog	This example is the catalog of Leeds Sports Pose Dataset, which is a dataset with *Keypoints2D* label.
NightOwls Catalog	This example is the catalog of NightOwls Dataset, which is a dataset with tracking *Box2D* label.

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

1.21.4 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`.

1.21.5 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`.

1.22 Label Format

TensorBay supports multiple types of labels.

Each `Data` instance 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 1.7: supported label types

supported label types	label classes	subcatalog classes
Classification	`Classification`	`ClassificationSubcatalog`
Box2D	`LabeledBox2D`	`Box2DSubcatalog`
Box3D	`LabeledBox3D`	`Box3DSubcatalog`
Keypoints2D	`LabeledKeypoints2D`	`Keypoints2DSubcatalog`
Polygon	`LabeledPolygon`	`PolygonSubcatalog`
MultiPolygon	`LabeledMultiPolygon`	`MultiPolygonSubcatalog`
RLE	`LabeledRLE`	`RLESubcatalog`
Polyline2D	`LabeledPolyline2D`	`Polyline2DSubcatalog`
MultiPolyline2D	`LabeledMultiPolyline2D`	`MultiPolyline2DSubcatalog`
Sentence	`LabeledSentence`	`SentenceSubcatalog`
SemanticMask	`SemanticMask`	`SemanticMaskSubcatalog`
InstanceMask	`InstanceMask`	`InstanceMaskSubcatalog`
PanopticMask	`PanopticMask`	`PanopticMaskSubcatalog`

1.22.1 Common Label Properties

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

Three common properties of a label will be introduced first, and the unique ones will be explained under the corresponding type of label.

Take a *2D box label* as an example:

```
>>> from tensorbay.label import LabeledBox2D
>>> box2d_label = LabeledBox2D(
... 10, 20, 30, 40,
... category="category",
```

(continues on next page)

(continued from previous page)

```
... attributes={"attribute_name": "attribute_value"},
... instance="instance_ID"
... )
>>> box2d_label
LabeledBox2D(10, 20, 30, 40)(
  (category): 'category',
  (attributes): {...},
  (instance): 'instance_ID'
)
```

category

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

```
>>> box2d_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.

```
>>> box2d_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.

```
>>> box2d_label.instance
"instance_ID"
```

1.22.2 Common Subcatalog Properties

Before creating a label or adding a label to data, it's necessary to define the annotation rules of the specific label type inside the dataset. This task is done by subcatalog.

Different label types have different subcatalog classes.

Take *Box2DSubcatalog* as an example to describe some common features of subcatalog.

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

tracking information

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.

Pass True to the is_tracking parameter while creating the subcatalog, or set the is_tracking attr after initialization.

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

category information

common category information

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.

Common category information can be added to the most subcatalogs except for mask subcatalogs.

```
>>> box2d_subcatalog.add_category(name="cat", description="The Flerken")
>>> box2d_subcatalog.categories
NameList [
  CategoryInfo("cat")
]
```

CategoryInfo is used to describe a *category*. See details in CategoryInfo.

mask category information

If the mask label in the dataset has category information, then the subcatalog should contain all the optional mask categories.

MaskCategory information can be added to the mask subcatalog.

Different from common category, mask category information must have category_id which is the pixel value of this category in all mask images.

```
>>> semantic_mask_subcatalog.add_category(name="cat", category_id=1, description="Ragdoll
↪")
>>> semantic_mask_subcatalog.categories
NameList [
  MaskCategoryInfo("cat")(...)
]
```

MaskCategoryInfo is used to describe the category information of pixels in the mask image. See details in MaskCategoryInfo.

attributes information

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

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

Attribute information ca be added to the subcatalog.

```
>>> box2d_subcatalog.add_attribute(
... name="attribute_name",
... type_="number",
... maximum=100,
... minimum=0,
... description="attribute description"
... )
>>> box2d_subcatalog.attributes
NameList [
  AttributeInfo("attribute_name")(...)
]
```

`AttributeInfo` is used to describe the rules of an *attributes*, which refers to the Json schema method.

See details in `AttributeInfo`.

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

1.22.3 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',
```

(continues on next page)

(continued from previous page)

```
  (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 *common category information* and *attributes information* for details.

The catalog with only Classification subcatalog is typically stored in a json file as follows:

```
{
    "CLASSIFICATION": {                                     <object>*
        "description":                                      <string>! -- Subcatalog␣
↪description, (default: "").
        "categoryDelimiter":                                <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                        Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                                        field by default␣
↪which means the category is of one level.
        "categories": [                                      <array>  -- Category list, which␣
↪contains all category information.
            {
                "name":                                     <string>* -- Category name.
                "description":                              <string>! -- Category description,␣
↪(default: "").
            },
            ...
            ...
        ],
        "attributes": [                                      <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                                     <string>* -- Attribute name.
                "enum": [...],                               <array>  -- All possible options␣
↪for the attribute.
                "type":                            <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
```

(continues on next page)

(continued from previous page)

```
                                                                   "number", "string",
↪"array" and "null". And it is not
                                                                   required when "enum"␣
↪is provided.
                "minimum":                                <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                                <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                                <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                         <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                       <string or array>  -- Type of elements in␣
↪the attribute array.
                    "minimum":                            <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                   "number".
                    "maximum":                            <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                   "number".
                },
                "parentCategories": [...],                 <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                   add this field if it␣
↪is a global attribute.
                "description":                            <string>! -- Attribute description,
↪ (default: "").
            },
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

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.

1.22.4 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 labels.

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, use the coordinates of the top-left and bottom-right vertexes of the 2D bounding box, or 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.from_xywh(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 *common category information*, *attributes information* and *tracking information* for details.

The catalog with only Box2D subcatalog is typically stored in a json file as follows:

```
{
    "BOX2D": {                                          <object>*
        "description":                                  <string>! -- Subcatalog␣
↪description, (default: "").
        "isTracking":                                  <boolean>! -- Whether this type of␣
↪label in the dataset contains tracking
                                                                     information,␣
↪(default: false).
        "categoryDelimiter":                            <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                     Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                                     field by default␣
↪which means the category is of one level.
```

(continues on next page)

(continued from previous page)

```
        "categories": [                                    <array>  -- Category list, which␣
↪contains all category information.
            {
                "name":                                    <string>* -- Category name.
                "description":                             <string>! -- Category description,␣
↪(default: "").
            },
            ...
            ...
        ],
        "attributes": [                                    <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                                    <string>* -- Attribute name.
                "enum": [...],                             <array>  -- All possible options␣
↪for the attribute.
                "type":                           <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                                     "number", "string",
↪"array" and "null". And it is not
                                                                     required when "enum"␣
↪is provided.
                "minimum":                                 <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                                 <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                                 <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                         <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                       <string or array>  -- Type of elements in␣
↪the attribute array.
                    "minimum":                             <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                     "number".
                    "maximum":                             <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                     "number".
                },
                "parentCategories": [...],                 <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                     add this field if it␣
↪is a global attribute.
                "description":                             <string>! -- Attribute description,
↪ (default: "").
            },
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

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.

1.22.5 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>
        }
    },
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    },
    "instance": <str>
}
```

To create a *LabeledBox3D* label:

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

Box3D.box3d

LabeledBox3D extends *Box3D*.

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

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

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)
```

(continues on next page)

(continued from previous page)

```
)
>>> 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 Box3D labels to data, `Box3DSubcatalog` should be defined.

`Box3DSubcatalog` has categories, attributes and tracking information, see *common category information*, *attributes information* and *tracking information* for details.

The catalog with only Box3D subcatalog is typically stored in a json file as follows:

```
{
    "BOX3D": {                                           <object>*
        "description":                                   <string>! -- Subcatalog␣
↪description, (default: "").
        "isTracking":                                   <boolean>! -- Whether this type of␣
↪label in the dataset contains tracking
                                                                    information,␣
↪(default: false).
        "categoryDelimiter":                             <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                    Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                                    field by default␣
↪which means the category is of one level.
        "categories": [                                   <array>  -- Category list, which␣
↪contains all category information.
            {
                "name":                                  <string>* -- Category name.
```

(continues on next page)

(continued from previous page)

```
                "description":                       <string>! -- Category description,␣
↪(default: "").
            },
            ...
            ...
        ],
        "attributes": [                              <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                              <string>* -- Attribute name.
                "enum": [...],                       <array>  -- All possible options␣
↪for the attribute.
                "type":                    <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                                 "number", "string",
↪"array" and "null". And it is not
                                                                 required when "enum"␣
↪is provided.
                "minimum":                           <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                           <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                           <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                   <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                <string or array>  -- Type of elements in␣
↪the attribute array.
                    "minimum":                       <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                 "number".
                    "maximum":                       <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                 "number".
                },
                "parentCategories": [...],           <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                 add this field if it␣
↪is a global attribute.
                "description":                       <string>! -- Attribute description,
↪ (default: "").
            },
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

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.

1.22.6 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): {...},
```

(continues on next page)

(continued from previous page)

```
  (instance): 'instance_ID'
)
```

Keypoints2D.keypoints2d

LabeledKeypoints2D extends *Keypoints2D*.

To construct a *LabeledKeypoints2D* instance with only the geometry information, The coordinates of the set of 2D keypoints are necessary. The visible status of each 2D keypoint is optional.

```
>>> 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, which can be obtained by index.

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

Keypoints2D.category

The category of the object inside the 2D keypoints. 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 2D keypoints, 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 *attributes information*, *common category information*, *tracking information* in *Keypoints2DSubcatalog*, it also has *keypoints* to describe a set of keypoints corresponding to certain categories.

The catalog with only Keypoints2D subcatalog is typically stored in a json file as follows:

```
{
    "KEYPOINTS2D": {                                        <object>*
        "description":                                      <string>! -- Subcatalog␣
↪description, (default: "").
        "isTracking":                                      <boolean>! -- Whether this type of␣
↪label in the dataset contains tracking
                                                                        information,␣
↪(default: false).
        "keypoints": [
            {
                "number":                                  <integer>* -- The number of key␣
↪points.
                "name":                                     <array>  -- The name of each key␣
↪point that corresponds to the
                                                                        "keypoints2d" in the␣
↪Keypoints2D label via index.
                "skeleton": [                               <array>  -- Key points skeleton␣
↪for visualization.
                    [<index>, <index>],                     <array>  -- Each array represents␣
↪a line segment. The skeleton is formed
                                                                        by connecting these␣
↪lines corresponding to the value of
                                                                        <index>.
                    ...
                ],
                "visible":                                  <string>  -- Indicates the meaning␣
↪of field "v" in the Keypoints2D label.
                                                                        There are two cases␣
↪as follows:
                                                                        1. "TERNARY": v=0:␣
↪invisible, v=1: occluded, v=2: visible.
                                                                        2. "BINARY": v=0:␣
↪invisible, v=1: visible.
                                                                        Do not add this field␣
↪if the field "v" does not exist.
                "parentCategories": [...],                  <array>  -- A list of categories␣
↪indicating to which category the
                                                                        keypoints rule␣
↪applies.Do not add this field if the keypoints
                                                                        rule applies to all␣
↪the categories of the entire dataset.
                "description":                              <string>! -- Key points␣
↪description, (default: "").
            },
        ],
```

(continues on next page)

(continued from previous page)

```
        "categoryDelimiter":                          <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                               Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                               field by default␣
↪which means the category is of one level.
        "categories": [                                <array>  -- Category list, which␣
↪contains all category information.
            {
                "name":                               <string>* -- Category name.
                "description":                        <string>! -- Category description,␣
↪(default: "").
            },
            ...
            ...
        ],
        "attributes": [                                <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                               <string>* -- Attribute name.
                "enum": [...],                         <array>  -- All possible options␣
↪for the attribute.
                "type":                      <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                               "number", "string",
↪"array" and "null". And it is not
                                                               required when "enum"␣
↪is provided.
                "minimum":                            <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                            <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                            <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                     <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                  <string or array>  -- Type of elements in␣
↪the attribute array.
                    "minimum":                        <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                               "number".
                    "maximum":                        <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                               "number".
                },
                "parentCategories": [...],             <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                               add this field if it␣
↪is a global attribute.
                "description":                        <string>! -- Attribute description,
↪ (default: "").
            },
```

(continues on next page)

(continued from previous page)

```
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

Besides giving the parameters while initializing *Keypoints2DSubcatalog*, it's also feasible to set them after initialization.

```
>>> from tensorbay.label import Keypoints2DSubcatalog
>>> keypoints2d_subcatalog = Keypoints2DSubcatalog()
>>> keypoints2d_subcatalog.add_keypoints(
... 3,
... names=["head", "body", "feet"],
... skeleton=[[0, 1], [1, 2]],
... visible="BINARY",
... parent_categories=["cat"],
... description="keypoints of cats"
... )
>>> keypoints2d_subcatalog.keypoints
[KeypointsInfo(
   (number): 3,
   (names): [...],
   (skeleton): [...],
   (visible): 'BINARY',
   (parent_categories): [...]
 )]
```

KeypointsInfo is used to describe a set of 2D keypoints.

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.

1.22.7 Polygon

Polygon is a type of label with a polygonal region on an image which contains some semantic information. It's often used for CV tasks such as semantic segmentation.

Each data can be assigned with multiple Polygon labels.

The structure of one Polygon label is like:

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

To create a *LabeledPolygon* label:

```
>>> from tensorbay.label import LabeledPolygon
>>> polygon_label = LabeledPolygon(
... [(1, 2), (2, 3), (1, 3)],
... category="category",
... attributes={"attribute_name": "attribute_value"},
... instance="instance_ID"
... )
>>> polygon_label
LabeledPolygon [
  Vector2D(1, 2),
  Vector2D(2, 3),
  Vector2D(1, 3)
](
  (category): 'category',
  (attributes): {...},
  (instance): 'instance_ID'
)
```

Polygon.polygon

LabeledPolygon extends *Polygon*.

To construct a *LabeledPolygon* instance with only the geometry information, use the coordinates of the vertexes of the polygonal region.

```
>>> LabeledPolygon([(1, 2), (2, 3), (1, 3)])
LabeledPolygon [
  Vector2D(1, 2),
  Vector2D(2, 3),
  Vector2D(1, 3)
]()
```

It contains the basic geometry information of the polygonal region.

```
>>> polygon_label.area()
0.5
```

Polygon.category

The category of the object inside the polygonal region. See *category* for details.

Polygon.attributes

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

Polygon.instance

Instance is the unique id for the object inside of the polygonal region, which is mostly used for tracking tasks. See *instance* for details.

PolygonSubcatalog

Before adding the Polygon labels to data, *PolygonSubcatalog* should be defined.

PolygonSubcatalog has categories, attributes and tracking information, see *common category information*, *attributes information* and *tracking information* for details.

The catalog with only Polygon subcatalog is typically stored in a json file as follows:

```
{
    "POLYGON": {                                         <object>*
        "description":                                   <string>! -- Subcatalog␣
↪description, (default: "").
        "isTracking":                                   <boolean>! -- Whether this type of␣
↪label in the dataset contains tracking
                                                                   information,␣
↪(default: false).
        "categoryDelimiter":                             <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                   Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
```

(continues on next page)

(continued from previous page)

```
                                                                  field by default␣
↪which means the category is of one level.
      "categories": [                               <array>  -- Category list, which␣
↪contains all category information.
          {
              "name":                               <string>* -- Category name.
              "description":                        <string>! -- Category description,␣
↪(default: "").
          },
          ...
          ...
      ],
      "attributes": [                               <array>  -- Attribute list, which␣
↪contains all attribute information.
          {
              "name":                               <string>* -- Attribute name.
              "enum": [...],                        <array>  -- All possible options␣
↪for the attribute.
              "type":                     <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                                  "number", "string",
↪"array" and "null". And it is not
                                                                  required when "enum"␣
↪is provided.
              "minimum":                            <number>  -- Minimum value of the␣
↪attribute when type is "number".
              "maximum":                            <number>  -- Maximum value of the␣
↪attribute when type is "number".
              "items": {                            <object>  -- Used only if the␣
↪attribute type is "array".
                  "enum": [...],                    <array>  -- All possible options␣
↪for elements in the attribute array.
                  "type":                 <string or array>  -- Type of elements in␣
↪the attribute array.
                  "minimum":                        <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                  "number".
                  "maximum":                        <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                  "number".
              },
              "parentCategories": [...],            <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                  add this field if it␣
↪is a global attribute.
              "description":                        <string>! -- Attribute description,
↪ (default: "").
          },
          ...
          ...
      ]
  }
```

(continues on next page)

(continued from previous page)

```
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

To add a *LabeledPolygon* label to one data:

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

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

1.22.8 MultiPolygon

MultiPolygon is a type of label with several polygonal regions which contain same semantic information on an image. It's often used for CV tasks such as semantic segmentation.

Each data can be assigned with multiple MultiPolygon labels.

The structure of one MultiPolygon label is like:

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

To create a *LabeledMultiPolygon* label:

```
>>> from tensorbay.label import LabeledMultiPolygon
>>> multipolygon_label = LabeledMultiPolygon(
... [[(1.0, 2.0), (2.0, 3.0), (1.0, 3.0)], [(1.0, 4.0), (2.0, 3.0), (1.0, 8.0)]],
```

(continues on next page)

(continued from previous page)

```
... category="category",
... attributes={"attribute_name": "attribute_value"},
... instance="instance_ID"
... )
>>> multipolygon_label
LabeledMultiPolygon [
  Polygon [...],
  Polygon [...]
](
  (category): 'category',
  (attributes): {...},
  (instance): 'instance_ID'
)
```

MultiPolygon.multi_polygon

LabeledMultiPolygon extends *MultiPolygon*.

To construct a *LabeledMultiPolygon* instance with only the geometry information, use the coordinates of the vertexes of polygonal regions.

```
>>> LabeledMultiPolygon([[[1.0, 4.0], [2.0, 3.7], [7.0, 4.0]],
... [[5.0, 7.0], [6.0, 7.0], [9.0, 8.0]]])
LabeledMultiPolygon [
  Polygon [...],
  Polygon [...]
]()
```

MultiPolygon.category

The category of the object inside polygonal regions. See *category* for details.

MultiPolygon.attributes

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

MultiPolygon.instance

Instance is the unique id for the object inside of polygonal regions, which is mostly used for tracking tasks. See *instance* for details.

MultiPolygonSubcatalog

Before adding the MultiPolygon labels to data, *MultiPolygonSubcatalog* should be defined.

MultiPolygonSubcatalog has categories, attributes and tracking information, see *common category information*, *attributes information* and *tracking information* for details.

The catalog with only MultiPolygon subcatalog is typically stored in a json file as follows:

```
{
    "MULTI_POLYGON": {                                          <object>*
        "description":                                          <string>! -- Subcatalog␣
↪description, (default: "").
        "isTracking":                                          <boolean>! -- Whether this type of␣
↪label in the dataset contains tracking
                                                                             information,␣
↪(default: false).
        "categoryDelimiter":                                    <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                             Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                                             field by default␣
↪which means the category is of one level.
        "categories": [                                          <array>  -- Category list, which␣
↪contains all category information.
            {
                "name":                                         <string>* -- Category name.
                "description":                                  <string>! -- Category description,␣
↪(default: "").
            },
            ...
            ...
        ],
        "attributes": [                                          <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                                         <string>* -- Attribute name.
                "enum": [...],                                   <array>  -- All possible options␣
↪for the attribute.
                "type":                                <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                                             "number", "string",
↪"array" and "null". And it is not
                                                                             required when "enum"␣
↪is provided.
                "minimum":                                      <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                                      <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                                      <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                               <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                            <string or array>  -- Type of elements in␣
↪the attribute array.
```

(continues on next page)

(continued from previous page)

```
                    "minimum":                          <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                  "number".
                    "maximum":                          <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                  "number".
                },
                "parentCategories": [...],               <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                  add this field if it␣
↪is a global attribute.
                "description":                          <string>! -- Attribute description,
↪ (default: "").
            },
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

To add a *LabeledMultiPolygon* label to one data:

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

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

1.22.9 RLE

RLE, Run-Length Encoding, is a type of label with a list of numbers to indicate whether the pixels are in the target region. It's often used for CV tasks such as semantic segmentation.

Each data can be assigned with multiple RLE labels.

The structure of one RLE label is like:

```
{
    "rle": [
        int,
        ...
    ]
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
```

(continues on next page)

(continued from previous page)

```
        ...
    }
    "instance": <str>
}
```

To create a *LabeledRLE* label:

```
>>> from tensorbay.label import LabeledRLE
>>> rle_label = LabeledRLE(
... [8, 4, 1, 3, 12, 7, 16, 2, 9, 2],
... category="category",
... attributes={"attribute_name": "attribute_value"},
... instance="instance_ID"
... )
>>> rle_label
LabeledRLE [
  8,
  4,
  1,
  ...
](
  (category): 'category',
  (attributes): {...},
  (instance): 'instance_ID'
)
```

RLE.rle

LabeledRLE extends *RLE*.

To construct a *LabeledRLE* instance with only the rle format mask.

```
>>> LabeledRLE([8, 4, 1, 3, 12, 7, 16, 2, 9, 2])
LabeledRLE [
  8,
  4,
  1,
  ...
]()
```

RLE.category

The category of the object inside the region represented by rle format mask. See *category* for details.

RLE.attributes

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

RLE.instance

Instance is the unique id for the object inside the region represented by rle format mask, which is mostly used for tracking tasks. See *instance* for details.

RLESubcatalog

Before adding the RLE labels to data, *RLESubcatalog* should be defined.

RLESubcatalog has categories, attributes and tracking information, see *common category information*, *attributes information* and *tracking information* for details.

The catalog with only RLE subcatalog is typically stored in a json file as follows:

```
{
    "RLE": {                                                <object>*
        "description":                                      <string>! -- Subcatalog␣
↪description, (default: "").
        "isTracking":                                      <boolean>! -- Whether this type of␣
↪label in the dataset contains tracking
                                                                         information,␣
↪(default: false).
        "categoryDelimiter":                                <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                         Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                                         field by default␣
↪which means the category is of one level.
        "categories": [                                      <array>  -- Category list, which␣
↪contains all category information.
            {
                "name":                                     <string>* -- Category name.
                "description":                              <string>! -- Category description,␣
↪(default: "").
            },
            ...
            ...
        ],
        "attributes": [                                      <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                                     <string>* -- Attribute name.
                "enum": [...],                               <array>  -- All possible options␣
↪for the attribute.
                "type":                            <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                                         "number", "string",
↪"array" and "null". And it is not
                                                                         required when "enum"␣
↪is provided.
```

(continues on next page)

(continued from previous page)

```
                "minimum":                          <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                          <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                          <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                   <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                <string or array>  -- Type of elements in␣
↪the attribute array.
                    "minimum":                      <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                 "number".
                    "maximum":                      <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                 "number".
                },
                "parentCategories": [...],           <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                 add this field if it␣
↪is a global attribute.
                "description":                      <string>! -- Attribute description,
↪ (default: "").
            },
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

To add a *LabeledRLE* label to one data:

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

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

1.22.10 Polyline2D

Polyline2D is a type of label with a 2D polyline on an image. It's often used for CV tasks such as lane detection.

Each data can be assigned with multiple Polyline2D labels.

The structure of one Polyline2D label is like:

```
{
    "polyline2d": [
        {
            "x": <float>
            "y": <float>
        },
        ...
        ...
    ],
    "beizerPointTypes": <str>
    "category": <str>
    "attributes": {
        <key>: <value>
        ...
        ...
    }
    "instance": <str>
}
```

Note: When the `is_beizer_curve` is `True` in the *Polyline2DSubcatalog*, `beizerPointTypes` is mandatory, where each character in the string represents the type of the point ("L" represents the vertex and "C" represents the control point) at the corresponding position in the `polyline2d` list.

To create a *LabeledPolyline2D* label:

```
>>> from tensorbay.label import LabeledPolyline2D
>>> polyline2d_label = LabeledPolyline2D(
... [(1, 2), (2, 3)],
... beizer_point_types="LL",
... category="category",
... attributes={"attribute_name": "attribute_value"},
... instance="instance_ID"
... )
>>> polyline2d_label
LabeledPolyline2D [
  Vector2D(1, 2),
  Vector2D(2, 3)
](
  (beizer_point_types): 'LL',
  (category): 'category',
  (attributes): {...},
  (instance): 'instance_ID'
)
```

Polyline2D.polyline2d

LabeledPolyline2D extends *Polyline2D*.

To construct a *LabeledPolyline2D* instance with only the geometry information, use the coordinates of the vertexes of the polyline.

```
>>> LabeledPolyline2D([[1, 2], [2, 3]])
LabeledPolyline2D [
  Vector2D(1, 2),
  Vector2D(2, 3)
]()
```

It contains a series of methods to operate on polyline.

```
>>> polyline_1 = LabeledPolyline2D([[1, 1], [1, 2], [2, 2]])
>>> polyline_2 = LabeledPolyline2D([[4, 5], [2, 1], [3, 3]])
>>> LabeledPolyline2D.uniform_frechet_distance(polyline_1, polyline_2)
3.6055512754639896
>>> LabeledPolyline2D.similarity(polyline_1, polyline_2)
0.2788897449072021
```

Polyline2D.category

The category of the 2D polyline. See *category* for details.

Polyline2D.attributes

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

Polyline2D.instance

Instance is the unique ID for the 2D polyline, which is mostly used for tracking tasks. See *instance* for details.

Polyline2DSubcatalog

Before adding the Polyline2D labels to data, *Polyline2DSubcatalog* should be defined.

Besides *common category information*, *attributes information* and *tracking information* in *Polyline2DSubcatalog*, it also has *is_beizer_curve* to describe the type of the polyline.

The catalog with only Polyline2D subcatalog is typically stored in a json file as follows:

```
{
    "POLYLINE2D": {                                         <object>*
        "description":                                      <string>! -- Subcatalog␣
↪description, (default: "").
        "isTracking":                                      <boolean>! -- Whether this type of␣
↪label in the dataset contains tracking
                                                                        information,␣
↪(default: false).
        "isBeizerCurve"                                    <boolean>! -- Whether the polyline␣
↪is a Bezier curve, (default: false).
```

(continues on next page)

(continued from previous page)

```
        "categoryDelimiter":                              <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                      Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                                      field by default␣
↪which means the category is of one level.
        "categories": [                                    <array>  -- Category list, which␣
↪contains all category information.
            {
                "name":                                   <string>* -- Category name.
                "description":                            <string>! -- Category description,␣
↪(default: "").
            },
            ...
            ...
        ],
        "attributes": [                                    <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                                   <string>* -- Attribute name.
                "enum": [...],                             <array>  -- All possible options␣
↪for the attribute.
                "type":                          <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                                      "number", "string",
↪"array" and "null". And it is not
                                                                      required when "enum"␣
↪is provided.
                "minimum":                                <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                                <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                                <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                         <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                      <string or array>  -- Type of elements in␣
↪the attribute array.
                    "minimum":                            <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                      "number".
                    "maximum":                            <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                      "number".
                },
                "parentCategories": [...],                 <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                      add this field if it␣
↪is a global attribute.
                "description":                            <string>! -- Attribute description,
↪ (default: "").
            },
```

(continues on next page)

(continued from previous page)

```
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

Besides giving the parameters while initializing `Polyline2DSubcatalog`, it's also feasible to set them after initialization.

```
>>> from tensorbay.label import Polyline2DSubcatalog
>>> polyline2d_subcatalog = Polyline2DSubcatalog()
>>> polyline2d_subcatalog.is_beizer_curve = True
>>> polyline2d_subcatalog
Polyline2DSubcatalog(
  (is_beizer_curve): True,
  (is_tracking): False
)
```

To add a *LabeledPolyline2D* label to one data:

```
>>> from tensorbay.dataset import Data
>>> data = Data("local_path")
>>> data.label.polyline2d = []
>>> data.label.polyline2d.append(polyline2d_label)
```

Note: One data may contain multiple Polyline2D labels, so the `Data.label.polyline2d` must be a list.

1.22.11 MultiPolyline2D

MultiPolyline2D is a type of label with several 2D polylines which belong to the same category on an image. It's often used for CV tasks such as lane detection.

Each data can be assigned with multiple MultiPolyline2D labels.

The structure of one MultiPolyline2D label is like:

```
{
    "multiPolyline2d": [
        [
            {
                "x": <float>
                "y": <float>
            },
            ...
            ...
        ],
        ...
```

(continues on next page)

(continued from previous page)

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

To create a *LabeledMultiPolyline2D* label:

```
>>> from tensorbay.label import LabeledMultiPolyline2D
>>> multipolyline2d_label = LabeledMultiPolyline2D(
... [[[1, 2], [2, 3]], [[3, 4], [6, 8]]],
... category="category",
... attributes={"attribute_name": "attribute_value"},
... instance="instance_ID"
... )
>>> multipolyline2d_label
LabeledMultiPolyline2D [
  Polyline2D [...],
  Polyline2D [...]
](
  (category): 'category',
  (attributes): {...},
  (instance): 'instance_ID'
)
```

MultiPolyline2D.multi_polyline2d

LabeledMultiPolyline2D extends *MultiPolyline2D*.

To construct a *LabeledMultiPolyline2D* instance with only the geometry information, use the coordinates of the vertexes of polylines.

```
>>> LabeledMultiPolyline2D([[[1, 2], [2, 3]], [[3, 4], [6, 8]]])
LabeledMultiPolyline2D [
  Polyline2D [...],
  Polyline2D [...]
]()
```

MultiPolyline2D.category

The category of the multiple 2D polylines. See *category* for details.

MultiPolyline2D.attributes

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

MultiPolyline2D.instance

Instance is the unique ID for the multiple 2D polylines, which is mostly used for tracking tasks. See *instance* for details.

MultiPolyline2DSubcatalog

Before adding the MultiPolyline2D labels to data, `MultiPolyline2DSubcatalog` should be defined.

`MultiPolyline2DSubcatalog` has categories, attributes and tracking information, see *common category information*, *attributes information* and *tracking information* for details.

The catalog with only MultiPolyline2D subcatalog is typically stored in a json file as follows:

```
{
    "MULTI_POLYLINE2D": {                                    <object>*
        "description":                                       <string>! -- Subcatalog␣
↪description, (default: "").
        "isTracking":                                       <boolean>! -- Whether this type of␣
↪label in the dataset contains tracking
                                                                          information,␣
↪(default: false).
        "categoryDelimiter":                                 <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                          Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                                          field by default␣
↪which means the category is of one level.
        "categories": [                                       <array>  -- Category list, which␣
↪contains all category information.
            {
                "name":                                      <string>* -- Category name.
                "description":                               <string>! -- Category description,␣
↪(default: "").
            },
            ...
            ...
        ],
        "attributes": [                                       <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                                      <string>* -- Attribute name.
                "enum": [...],                                <array>  -- All possible options␣
↪for the attribute.
                "type":                             <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
```

(continues on next page)

(continued from previous page)

```
                                                                    "number", "string",
↪"array" and "null". And it is not
                                                                    required when "enum"␣
↪is provided.
                "minimum":                                <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                                <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                                <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                         <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                      <string or array>  -- Type of elements in␣
↪the attribute array.
                    "minimum":                            <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                    "number".
                    "maximum":                            <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                    "number".
                },
                "parentCategories": [...],                 <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                    add this field if it␣
↪is a global attribute.
                "description":                            <string>! -- Attribute description,
↪ (default: "").
            },
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

To add a *LabeledMultiPolyline2D* label to one data:

```
>>> from tensorbay.dataset import Data
>>> data = Data("local_path")
>>> data.label.multi_polyline2d = []
>>> data.label.multi_polyline2d.append(multipolyline2d_label)
```

Note: One data may contain multiple MultiPolyline2D labels, so the `Data.label.multi_polyline2d` must be a list.

1.22.12 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>
        }
        ...
        ...
    ],
    "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(
```

(continues on next page)

(continued from previous page)

```
  (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
    )
  ],
  (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,
```

(continues on next page)

(continued from previous page)

```
  (end): 2
)
```

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

Sentence.attributes

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

SentenceSubcatalog

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

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

The catalog with only Sentence subcatalog is typically stored in a json file as follows:

```
{
    "SENTENCE": {                                           <object>*
        "isSample":                                        <boolean>! -- Whether the unit of␣
↪sampling points in Sentence label is the
                                                                         number of samples.␣
↪The default value is false and the units
                                                                         are seconds.
        "sampleRate":                                       <number>  -- Audio sampling␣
↪frequency whose unit is Hz. It is required
                                                                         when "isSample" is␣
↪true.
        "description":                                      <string>! -- Subcatalog␣
↪description, (default: "").
        "attributes": [                                      <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                                     <string>* -- Attribute name.
                "enum": [...],                               <array>  -- All possible options␣
↪for the attribute.
                "type":                            <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                                         "number", "string",
↪"array" and "null". And it is not
                                                                         required when "enum"␣
↪is provided.
                "minimum":                                  <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                                  <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                                  <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                           <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                        <string or array>  -- Type of elements in␣
↪the attribute array.
```

(continues on next page)

(continued from previous page)

```
                    "minimum":                          <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                    "number".
                    "maximum":                          <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                    "number".
                },
                "description":                          <string>! -- Attribute description,
↪ (default: "").
            },
            ...
            ...
        ]
        "lexicon": [                                    <array>   -- A list consists all␣
↪of text and phone.
            [
                text,                                   <string>  -- Word.
                phone,                                  <string>  -- Corresponding␣
↪phonemes.
                phone,                                  <string>  -- Corresponding␣
↪phonemes (A word can correspond to more than
                                                                    one phoneme).
                ...
            ],
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

Besides giving the parameters while initializing *SentenceSubcatalog*, it's also feasible to set them after initialization.

```
>>> 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.

1.22.13 SemanticMask

SemanticMask is a type of label which is usually used for semantic segmentation task.

In TensorBay, the structure of SemanticMask label is unified as follows:

```
{
    "localPath": <str>
    "info": [
        {
            "categoryId": <int>
            "attributes": {
                <key>: <value>
                ...
                ...
            }
        },
        ...
        ...
    ]
}
```

`local_path` is the storage path of the mask image. TensorBay only supports single-channel, gray-scale png images. If the number of categories exceeds 256, the color depth of this image should be 16 bits, otherwise it is 8 bits.

The gray-scale value of the pixel corresponds to the category id of the `categories` within the *SemanticMaskSubcatalog*.

Each data can only be assigned with one *SemanticMask* label.

To create a *SemanticMask* label:

```
>>> from tensorbay.label import SemanticMask
>>> semantic_mask_label = SemanticMask(local_path="/semantic_mask/mask_image.png")
>>> semantic_mask_label
SemanticMask("/semantic_mask/mask_image.png")()
```

SemanticMask.all_attributes

`all_attributes` is a dictionary that stores attributes for each category. Each attribute is stored in key-value pairs. See *attributes* for details.

To create *all_attributes*:

```
>>> semantic_mask_label.all_attributes = {1: {"occluded": True}, 2: {"occluded": False}}
>>> semantic_mask_label
SemanticMask("/semantic_mask/mask_image.png")(
  (all_attributes): {
    1: {
      'occluded': True
```

(continues on next page)

(continued from previous page)

```
    },
    2: {
      'occluded': False
    }
  }
)
```

Note: In *SemanticMask*, the key of *all_attributes* is category id which should be an integer.

SemanticMaskSubcatalog

Before adding the SemanticMask labels to data, *SemanticMaskSubcatalog* should be defined.

SemanticMaskSubcatalog has mask categories and attributes, see *mask category information* and *attributes information* for details.

The catalog with only SemanticMask subcatalog is typically stored in a json file as follows:

```
{
    "SEMANTIC_MASK": {                                    <object>*
        "description":                                    <string>! -- Subcatalog␣
↪description, (default: "").
        "categoryDelimiter":                              <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                       Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                                       field by default␣
↪which means the category is of one level.
        "categories": [                                    <array>* -- Category list, which␣
↪contains all category information.
            {
                "name":                                   <string>* -- Category name.
                "categoryId":                            <integer>* -- Category id.
                "description":                            <string>! -- Category description,␣
↪(default: "").
            },
            ...
            ...
        ],
        "attributes": [                                    <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                                   <string>* -- Attribute name.
                "enum": [...],                             <array>  -- All possible options␣
↪for the attribute.
                "type":                          <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                                       "number", "string",
↪"array" and "null". And it is not
                                                                       required when "enum"␣
↪is provided.
```

(continues on next page)

(continued from previous page)

```
                "minimum":                          <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                          <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                          <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                   <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                <string or array>  -- Type of elements in␣
↪the attribute array.
                    "minimum":                      <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                 "number".
                    "maximum":                      <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                 "number".
                },
                "parentCategories": [...],           <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                 add this field if it␣
↪is a global attribute.
                "description":                      <string>! -- Attribute description,
↪ (default: "").
            },
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

To add a *SemanticMask* label to one data:

```
>>> from tensorbay.dataset import Data
>>> data = Data("local_path")
>>> data.label.semantic_mask = semantic_mask_label
```

Note: One data can only have one SemanticMask label, See `Data.label.semantic_mask` for details.

1.22.14 InstanceMask

InstanceMask is a type of label which is usually used for instance segmentation task.

In TensorBay, the structure of InstanceMask label is unified as follows:

```
{
    "localPath": <str>
    "info": [
        {
            "instanceId": <int>
            "attributes": {
                <key>: <value>
                ...
                ...
            }
        },
        ...
        ...
    ]
}
```

`local_path` is the storage path of the mask image. TensorBay only supports single-channel, gray-scale png images. If the number of categories exceeds 256, the color depth of this image should be 16 bits, otherwise it is 8 bits.

There are pixels in the InstanceMask that do not represent the instance, such as backgrounds or borders. This information is written to the `categories` within the *InstanceMaskSubcatalog*.

Each data can only be assigned with one *InstanceMask* label.

To create a *InstanceMask* label:

```
>>> from tensorbay.label import InstanceMask
>>> instance_mask_label = InstanceMask(local_path="/instance_mask/mask_image.png")
>>> instance_mask_label
InstanceMask("/instance_mask/mask_image.png")()
```

InstanceMask.all_attributes

all_attributes is a dictionary that stores attributes for each instance. Each attribute is stored in key-value pairs. See *attributes* for details.

To create *all_attributes*:

```
>>> instance_mask_label.all_attributes = {1: {"occluded": True}, 2: {"occluded": True}}
>>> instance_mask_label
InstanceMask("/instance_mask/mask_image.png")(
  (all_attributes): {
    1: {
      'occluded': True
    },
    2: {
      'occluded': True
    }
```

(continues on next page)

(continued from previous page)

```
    }
)
```

Note: In *InstanceMask*, the key of *all_attributes* is instance id which should be an integer.

InstanceMaskSubcatalog

Before adding the InstanceMask labels to data, *InstanceMaskSubcatalog* should be defined.

InstanceMaskSubcatalog has mask categories and attributes, see *mask category information* and *attributes information* for details.

The catalog with only InstanceMask subcatalog is typically stored in a json file as follows:

```
{
    "INSTANCE_MASK": {                                  <object>*
        "description":                                  <string>! -- Subcatalog␣
↪description, (default: "").
        "isTracking":                                  <boolean>! -- Whether this type of␣
↪label in the dataset contains tracking
                                                                     information,␣
↪(default: false).
        "categoryDelimiter":                            <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                     Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                                     field by default␣
↪which means the category is of one level.
        "categories": [                                  <array>  -- The categories of␣
↪pixels in the InstanceMask that do not
                                                                     represent the␣
↪instance, such as backgrounds or borders.
            {
                "name":                                 <string>* -- Category name.
                "categoryId":                          <integer>* -- Category id.
                "description":                          <string>! -- Category description,␣
↪(default: "").
            },
            ...
            ...
        ],
        "attributes": [                                  <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                                 <string>* -- Attribute name.
                "enum": [...],                           <array>  -- All possible options␣
↪for the attribute.
                "type":                        <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                                     "number", "string",
↪"array" and "null". And it is not
```

(continues on next page)

(continued from previous page)

```
                                                                    required when "enum"␣
↪is provided.
                "minimum":                              <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                              <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                              <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                       <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                     <string or array>  -- Type of elements in␣
↪the attribute array.
                    "minimum":                          <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                    "number".
                    "maximum":                          <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                    "number".
                },
                "parentCategories": [...],               <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                    add this field if it␣
↪is a global attribute.
                "description":                          <string>! -- Attribute description,
↪ (default: "").
            },
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

To add a *InstanceMask* label to one data:

```
>>> from tensorbay.dataset import Data
>>> data = Data("local_path")
>>> data.label.instance_mask = instance_mask_label
```

Note: One data can only have one InstanceMask label, See `Data.label.instance_mask` for details.

1.22.15 PanopticMask

PanopticMask is a type of label which is usually used for panoptic segmentation task.

In TensorBay, the structure of PanopticMask label is unified as follows:

```
{
    "localPath": <str>
    "info": [
        {
            "instanceId": <int>
            "categoryId": <int>
            "attributes": {
                <key>: <value>
                ...
                ...
            }
        }
        ...
        ...
    ],
}
```

`local_path` is the storage path of the mask image. TensorBay only supports single-channel, gray-scale png images. If the number of categories exceeds 256, the color depth of this image should be 16 bits, otherwise it is 8 bits.

The gray-scale value of the pixel corresponds to the category id of the `categories` within the *PanopticMaskSubcatalog*.

Each data can only be assigned with one *PanopticMask* label.

To create a *PanopticMask* label:

```
>>> from tensorbay.label import PanopticMask
>>> panoptic_mask_label = PanopticMask(local_path="/panoptic_mask/mask_image.png")
>>> panoptic_mask_label.all_category_ids = {1: 2, 2: 2}
>>> panoptic_mask_label
PanopticMask("/panoptic_mask/mask_image.png")(
  (all_category_ids): {
    1: 2,
    2: 2
  }
)
```

Note: In *PanopticMask*, the key and value of *all_category_ids* are instance id and category id, respectively, which both should be integers.

PanopticMask.all_attributes

all_attributes is a dictionary that stores attributes for each instance. Each attribute is stored in key-value pairs. See *attributes* for details.

To create *all_attributes*:

```
>>> panoptic_mask_label.all_attributes = {1: {"occluded": True}, 2: {"occluded": True}}
>>> panoptic_mask_label
PanopticMask("/panoptic_mask/mask_image.png")(
  (all_category_ids): {
    1: 2,
    2: 2
  },
  (all_attributes): {
    1: {
      'occluded': True
    },
    2: {
      'occluded': True
    }
  }
)
```

Note: In `PanopticMask`, the key of *all_attributes* is instance id which should be integer.

PanopticMaskSubcatalog

Before adding the PanopticMask labels to data, `PanopticMaskSubcatalog` should be defined.

`PanopticMaskSubcatalog` has mask categories and attributes, see *mask category information* and *attributes information* for details.

The catalog with only PanopticMask subcatalog is typically stored in a json file as follows:

```
{
    "PANOPTIC_MASK": {                                      <object>*
        "description":                                      <string>! -- Subcatalog␣
↪description, (default: "").
        "categoryDelimiter":                                <string>  -- The delimiter in␣
↪category names indicating subcategories.
                                                                         Recommended delimiter␣
↪is ".". There is no "categoryDelimiter"
                                                                         field by default␣
↪which means the category is of one level.
        "categories": [                                      <array>* -- Category list, which␣
↪contains all category information.
            {
                "name":                                     <string>* -- Category name.
                "categoryId":                              <integer>* -- Category id.
                "description":                              <string>! -- Category description,␣
↪(default: "").
```

(continues on next page)

(continued from previous page)

```
            },
            ...
            ...
        ],
        "attributes": [                                 <array>  -- Attribute list, which␣
↪contains all attribute information.
            {
                "name":                                 <string>* -- Attribute name.
                "enum": [...],                          <array>  -- All possible options␣
↪for the attribute.
                "type":                        <string or array>  -- Type of the attribute␣
↪including "boolean", "integer",
                                                                    "number", "string",
↪"array" and "null". And it is not
                                                                    required when "enum"␣
↪is provided.
                "minimum":                              <number>  -- Minimum value of the␣
↪attribute when type is "number".
                "maximum":                              <number>  -- Maximum value of the␣
↪attribute when type is "number".
                "items": {                              <object>  -- Used only if the␣
↪attribute type is "array".
                    "enum": [...],                      <array>  -- All possible options␣
↪for elements in the attribute array.
                    "type":                    <string or array>  -- Type of elements in␣
↪the attribute array.
                    "minimum":                          <number>  -- Minimum value of␣
↪elements in the attribute array when type is
                                                                    "number".
                    "maximum":                          <number>  -- Maximum value of␣
↪elements in the attribute array when type is
                                                                    "number".
                },
                "parentCategories": [...],              <array>  -- Indicates the␣
↪category to which the attribute belongs. Do not
                                                                    add this field if it␣
↪is a global attribute.
                "description":                          <string>! -- Attribute description,
↪ (default: "").
            },
            ...
            ...
        ]
    }
}
```

Note: * indicates that the field is required. ! indicates that the field has a default value.

To add a *PanopticMask* label to one data:

```
>>> from tensorbay.dataset import Data
>>> data = Data("local_path")
>>> data.label.panoptic_mask = panoptic_mask_label
```

Note: One data can only have one PanopticMask label, See `Data.label.panoptic_mask` for details.

1.23 Exceptions

TensorBay SDK defines a series of custom exceptions.

1.23.1 Exception Hierarchy

The class hierarchy for TensorBay custom exceptions is:

```
+-- TensorBayException
    +-- ClientError
        +-- StatusError
        +-- DatasetTypeError
        +-- FrameError
        +-- ResponseError
            +-- AccessDeniedError
            +-- ForbiddenError
            +-- InvalidParamsError
            +-- NameConflictError
            +-- RequestParamsMissingError
            +-- ResourceNotExistError
            +-- InternalServerError
            +-- UnauthorizedError
    +-- TBRNError
    +-- OpenDatasetError
        +-- NoFileError
        +-- FileStructureError
```

1.23.2 Exception Definitions

The definitions for TensorBay custom exceptions are:

AccessDeniedError *`AccessDeniedError`* defines the exception for access denied response error in the client module. Raised when the current account has no permission to access the resource.

ClientError *`ClientError`* is the base class for custom exceptions in the client module.

DatasetTypeError *`DatasetTypeError`* defines the exception for incorrect type of the requested dataset in the client module. Raised when the type of the required dataset is inconsistent with the input "is_fusion" parameter while getting dataset from TensorBay.

FileStructureError *`FileStructureError`* defines the exception for incorrect file structure in the opendataset directory.

ForbiddenError *ForbiddenError* defines the exception for illegal operations Tensorbay forbids. Raised when the current operation is forbidden by Tensorbay.

FrameError *FrameError* defines the exception for incorrect frame id in the client module. Raised when the frame id and timestamp of a frame conflicts or missing.

InternalServerError *InternalServerError* defines the exception for internal server error in the client module. Raised when internal server error was responded.

InvalidParamsError *InvalidParamsError* defines the exception for invalid parameters response error in the client module. Raised when the parameters of the request are invalid.

NameConflictError *NameConflictError* defines the exception for name conflict response error in the client module. Raised when the name of the resource to be created already exists on Tensorbay.

NoFileError *NoFileError* defines the exception for no matching file found in the opendataset directory.

OpenDatasetError *OpenDatasetError* is the base class for custom exceptions in the opendataset module.

RequestParamsMissingError *RequestParamsMissingError* defines the exception for request parameters missing response error in the client module. Raised when necessary parameters of the request are missing.

ResourceNotExistError *ResourceNotExistError* defines the exception for resource not existing response error in the client module. Raised when the request resource does not exist on Tensorbay.

ResponseError *ResponseError* defines the exception for post response error in the client module. Raised when the response from TensorBay has error. And different subclass exceptions will be raised according to different error code.

StatusError *StatusError* defines the exception for illegal status in the client module. Raised when the status is draft or commit, while the required status is commit or draft.

TBRNError *TBRNError* defines the exception for invalid TBRN. Raised when the TBRN format is incorrect.

TensorBayException *TensorBayException* is the base class for TensorBay SDK custom exceptions.

UnauthorizedError *UnauthorizedError* defines the exception for unauthorized response error in the client module. Raised when the *accesskey* is incorrect.

1.24 API Reference

1.24.1 tensorbay.client

tensorbay.client.storage_config

Class CloudClient.

The `CloudClient` defines the initial client to interact between local and cloud platform.

class `tensorbay.client.cloud_storage.`**`CloudClient`**(*name*, *client*)

Bases: `object`

`CloudClient` defines the client to interact between local and cloud platform.

Parameters

- **name** (*str*) – Name of the auth cloud storage config.
- **client** (`tensorbay.client.requests.Client`) – The initial client to interact between local and TensorBay.

Return type None

list_auth_data(*path=''*)

List all cloud files in the given directory as `AuthData`.

Parameters **path** (*str*) – The directory path on the cloud platform.

Returns The list of `AuthData` of all the cloud files.

Return type List[*tensorbay.dataset.data.AuthData*]

class tensorbay.client.cloud_storage.**StorageConfig**(*name*, *file_path*, *type_*, *is_graviti_storage*)

Bases: `tensorbay.utility.attr.AttrsMixin`, `tensorbay.utility.repr.ReprMixin`

This is a class stores the information of storage config.

Parameters

- **name** (*str*) – The storage config name.
- **file_path** (*str*) – Storage config path of the bucket.
- **type** – Type of the storage provider, such as oss, s3, azure.
- **is_graviti_storage** (*bool*) – Whether the config is belong to graviti.
- **type_** (*str*) –

Return type None

classmethod loads(*contents*)

Loads a `StorageConfig` instance from the given contents.

Parameters **contents** (*Dict[str, Any]*) – The given dict containing the storage config:

```
{
    "name":                 <str>,
    "filePath":             <str>,
    "type":                 <str>,
    "isGravitiStorage":     <boolean>
}
```

Returns The loaded `StorageConfig` instance.

Return type tensorbay.client.cloud_storage._T

dumps()

Dumps the storage config into a dict.

Returns

A dict containing all the information of the StorageConfig:

```
{
    "name":                 <str>,
    "filePath":             <str>,
    "type":                 <str>,
    "isGravitiStorage":     <boolean>
}
```

Return type Dict[str, Any]

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.**DatasetClientBase**(*name*, *dataset_id*, *gas*, ***, *status*, *alias*, *is_public*)

Bases: `tensorbay.client.version.VersionControlMixin`

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** (*str*) – Dataset name.
- **dataset_id** (*str*) – Dataset ID.
- **gas** (GAS) – The initial client to interact between local and TensorBay.
- **status** (`tensorbay.client.status.Status`) – The version control status of the dataset.
- **alias** (*str*) – Dataset alias.
- **is_public** (*bool*) – Whether the dataset is public.

Return type None

name

Dataset name.

dataset_id

Dataset ID.

status

The version control status of the dataset.

property is_public: bool

Return whether the dataset is public.

Returns Whether the dataset is public.

property cache_enabled: bool

Whether the cache is enabled.

Returns Whether the cache is enabled.

enable_cache(*cache_path=''*)

Enable cache when open the remote data of the dataset.

Parameters **cache_path** (*str*) – The path to store the cache.

Raises `StatusError` – When enable cache under draft status.

Return type None

`update_notes`(*, *is_continuous=None*, *bin_point_cloud_fields=Ellipsis*)
Update the notes.

Parameters

- **is_continuous** (`Optional[bool]`) – Whether the data is continuous.
- **bin_point_cloud_fields** (`Optional[Iterable[str]]`) – The field names of the bin point cloud files in the dataset.

Return type None

`get_notes`()
Get the notes.

Returns The `Notes`.

Return type *tensorbay.dataset.dataset.Notes*

`list_segment_names`()
List all segment names in a certain commit.

Returns The PagingList of segment names.

Return type *tensorbay.client.lazy.PagingList*[str]

`get_catalog`()
Get the catalog of the certain commit.

Returns Required `Catalog`.

Return type *tensorbay.label.catalog.Catalog*

`upload_catalog`(*catalog*)
Upload a catalog to the draft.

Parameters **catalog** (`tensorbay.label.catalog.Catalog`) – `Catalog` to upload.

Return type None

`delete_segment`(*name*)
Delete a segment of the draft.

Parameters **name** (`str`) – Segment name.

Return type None

`get_label_statistics`()
Get label statistics of the dataset.

Returns Required `Statistics`.

Return type *tensorbay.client.statistics.Statistics*

`get_total_size`()
Get total data size of the dataset and the unit is byte.

Returns The total data size of the dataset.

Return type int

class `tensorbay.client.dataset.DatasetClient`(*name*, *dataset_id*, *gas*, *, *status*, *alias*, *is_public*)
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.

Parameters

- **name** (*str*) –
- **dataset_id** (*str*) –
- **gas** (GAS) –
- **status** (tensorbay.client.status.Status) –
- **alias** (*str*) –
- **is_public** (*bool*) –

Return type None

get_or_create_segment(*name='default'*)

Get or create a segment with the given name.

Parameters **name** (*str*) – The name of the fusion segment.

Returns The created `SegmentClient` with given name.

Return type *tensorbay.client.segment.SegmentClient*

create_segment(*name='default'*)

Create a segment with the given name.

Parameters **name** (*str*) – The name of the fusion segment.

Returns The created `SegmentClient` with given name.

Raises **NameConflictError** – When the segment exists.

Return type *tensorbay.client.segment.SegmentClient*

copy_segment(*source_name*, *target_name=None*, ***, *source_client=None*, *strategy='abort'*)

Copy segment to this dataset.

Parameters

- **source_name** (*str*) – The source name of the copied segment.
- **target_name** (*Optional[str]*) – The target name of the copied segment. This argument is used to specify a new name of the copied segment. If None, the name of the copied segment will not be changed after copy.
- **source_client** (*Optional[*tensorbay.client.dataset.DatasetClient*]*) – The source dataset client of the copied segment. This argument is used to specify where the copied segment comes from when the copied segment is from another commit, draft or even another dataset. If None, the copied segment comes from this dataset.
- **strategy** (*str*) – The strategy of handling the name conflict. There are three options:

1. "abort": stop copying and raise exception;
2. "override": the source segment will override the origin segment;
3. "skip": keep the origin segment.

Returns The client of the copied target segment.

Return type *tensorbay.client.segment.SegmentClient*

move_segment(*source_name*, *target_name*, *, *strategy='abort'*)
Move/Rename segment in this dataset.

Parameters

- **source_name** (*str*) – The source name of the moved segment.
- **target_name** (*str*) – The target name of the moved segment.
- **strategy** (*str*) – The strategy of handling the name conflict. There are three options:

1. "abort": stop moving and raise exception;
2. "override": the source segment will override the origin segment;
3. "skip": keep the origin segment.

Returns The client of the moved target segment.

Return type *tensorbay.client.segment.SegmentClient*

get_segment(*name='default'*)
Get a segment in a certain commit according to given name.

Parameters **name** (*str*) – The name of the required segment.

Returns The required `SegmentClient`.

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

Return type *tensorbay.client.segment.SegmentClient*

upload_segment(*segment*, *, *jobs=1*, *skip_uploaded_files=False*, *quiet=False*, *_is_cli=False*)
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** (`tensorbay.dataset.segment.Segment`) – The `Segment` contains the information needs to be upload.
- **jobs** (*int*) – The number of the max workers in multi-thread uploading method.
- **skip_uploaded_files** (*bool*) – True for skipping the uploaded files.
- **quiet** (*bool*) – Set to True to stop showing the upload process bar.
- **_is_cli** (*bool*) – Whether the method is called by CLI.

Raises **Exception** – When the upload got interrupted by Exception.

Returns The `SegmentClient` used for uploading the data in the segment.

Return type *tensorbay.client.segment.SegmentClient*

get_diff(*, *head=None*)
Get a brief diff between head and its parent commit.

Parameters **head** (*Optional[Union[str, int]]*) – Target version identification. Type int for draft number, type str for revision. If not given, use the current commit id.

Return type *tensorbay.client.diff.DatasetDiff*

Examples

```
>>> self.get_diff(head="b382450220a64ca9b514dcef27c82d9a")
```

Returns The brief diff between head and its parent commit.

Parameters **head** (*Optional[Union[str, int]]*) –

Return type *tensorbay.client.diff.DatasetDiff*

class tensorbay.client.dataset.**FusionDatasetClient**(*name*, *dataset_id*, *gas*, *, *status*, *alias*, *is_public*)

Bases: `tensorbay.client.dataset.DatasetClientBase`

This class defines `FusionDatasetClient`.

`FusionDatasetClient` inherits from `DatasetClientBase` and provides more methods within a fusion dataset scope, such as `FusionDatasetClient.get_segment()`, `FusionDatasetClient.commit` and `FusionDatasetClient.upload_segment()`. In contrast to `DatasetClient`, a `FusionDatasetClient` has multiple sensors.

Parameters

- **name** (*str*) –
- **dataset_id** (*str*) –
- **gas** (GAS) –
- **status** (tensorbay.client.status.Status) –
- **alias** (*str*) –
- **is_public** (*bool*) –

Return type None

get_or_create_segment(*name='default'*)

Get or create a fusion segment with the given name.

Parameters **name** (*str*) – The name of the fusion segment.

Returns The created `FusionSegmentClient` with given name.

Return type *tensorbay.client.segment.FusionSegmentClient*

create_segment(*name='default'*)

Create a fusion segment with the given name.

Parameters **name** (*str*) – The name of the fusion segment.

Returns The created `FusionSegmentClient` with given name.

Raises **NameConflictError** – When the segment exists.

Return type *tensorbay.client.segment.FusionSegmentClient*

copy_segment(*source_name*, *target_name=None*, *, *source_client=None*, *strategy='abort'*)

Copy segment to this dataset.

Parameters

- **source_name** (*str*) – The source name of the copied segment.
- **target_name** (*Optional[str]*) – The target name of the copied segment. This argument is used to specify a new name of the copied segment. If None, the name of the copied segment will not be changed after copy.

- **source_client** (*Optional[tensorbay.client.dataset.FusionDatasetClient]*) – The source dataset client of the copied segment. This argument is used to specify where the copied segment comes from when the copied segment is from another commit, draft or even another dataset. If None, the copied segment comes from this dataset.
- **strategy** (*str*) – The strategy of handling the name conflict. There are three options:

1. "abort": stop copying and raise exception;
2. "override": the source segment will override the origin segment;
3. "skip": keep the origin segment.

Returns The client of the copied target segment.

Return type *tensorbay.client.segment.FusionSegmentClient*

move_segment(*source_name*, *target_name*, *, *strategy='abort'*)
Move/Rename segment in this dataset.

Parameters

- **source_name** (*str*) – The source name of the moved segment.
- **target_name** (*str*) – The target name of the moved segment.
- **strategy** (*str*) – The strategy of handling the name conflict. There are three options:

1. "abort": stop moving and raise exception;
2. "override": the source segment will override the origin segment;
3. "skip": keep the origin segment.

Returns The client of the moved target segment.

Return type *tensorbay.client.segment.FusionSegmentClient*

get_segment(*name='default'*)
Get a fusion segment in a certain commit according to given name.

Parameters **name** (*str*) – The name of the required fusion segment.

Returns The required *FusionSegmentClient*.

Raises ***ResourceNotExistError*** – When the required fusion segment does not exist.

Return type *tensorbay.client.segment.FusionSegmentClient*

upload_segment(*segment*, *, *jobs=1*, *skip_uploaded_files=False*, *quiet=False*)
Upload a fusion segment object to the draft.

This function will upload all info contains in the input *FusionSegment*, which includes:

- Create a segment using the name of input fusion segment object.
- Upload all sensors in the segment to the dataset.
- Upload all frames in the segment to the dataset.

Parameters

- **segment** (*tensorbay.dataset.segment.FusionSegment*) – The *FusionSegment*.
- **jobs** (*int*) – The number of the max workers in multi-thread upload.
- **skip_uploaded_files** (*bool*) – Set it to True to skip the uploaded files.

- **quiet** (*bool*) – Set to True to stop showing the upload process bar.

Raises **Exception** – When the upload got interrupted by Exception.

Returns

The `FusionSegmentClient` used for uploading the data in the segment.

Return type *tensorbay.client.segment.FusionSegmentClient*

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*, *url=''*)

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

Return type None

get_user()

Get the user information with the current accesskey.

Returns The `struct.UserInfo` with the current accesskey.

Return type *tensorbay.client.struct.UserInfo*

get_auth_storage_config(*name*)

Get the auth storage config with the given name.

Parameters **name** (*str*) – The required auth storage config name.

Returns The auth storage config with the given name.

Raises

- **TypeError** – When the given auth storage config is illegal.
- **ResourceNotExistError** – When the required auth storage config does not exist.

Return type *tensorbay.client.cloud_storage.StorageConfig*

list_auth_storage_configs()

List auth storage configs.

Returns The PagingList of all auth storage configs.

Return type *tensorbay.client.lazy.PagingList*[*tensorbay.client.cloud_storage.StorageConfig*]

delete_storage_config(*name*)

Delete a storage config in TensorBay.

Parameters **name** (*str*) – Name of the storage config, unique for a team.

Return type None

create_oss_storage_config(*name*, *file_path*, *, *endpoint*, *accesskey_id*, *accesskey_secret*, *bucket_name*)
Create an oss auth storage config.

Parameters

- **name** (*str*) – The required auth storage config name.
- **file_path** (*str*) – dataset storage path of the bucket.
- **endpoint** (*str*) – endpoint of the oss.
- **accesskey_id** (*str*) – accesskey_id of the oss.
- **accesskey_secret** (*str*) – accesskey_secret of the oss.
- **bucket_name** (*str*) – bucket_name of the oss.

Returns The cloud client of this dataset.

Return type *tensorbay.client.cloud_storage.CloudClient*

create_s3_storage_config(*name*, *file_path*, *, *endpoint*, *accesskey_id*, *accesskey_secret*, *bucket_name*)
Create a s3 auth storage config.

Parameters

- **name** (*str*) – The required auth storage config name.
- **file_path** (*str*) – dataset storage path of the bucket.
- **endpoint** (*str*) – endpoint of the s3.
- **accesskey_id** (*str*) – accesskey_id of the s3.
- **accesskey_secret** (*str*) – accesskey_secret of the s3.
- **bucket_name** (*str*) – bucket_name of the s3.

Returns The cloud client of this dataset.

Return type *tensorbay.client.cloud_storage.CloudClient*

create_azure_storage_config(*name*, *file_path*, *, *account_type*, *account_name*, *account_key*, *container_name*)
Create an azure auth storage config.

Parameters

- **name** (*str*) – The required auth storage config name.
- **file_path** (*str*) – dataset storage path of the bucket.
- **account_type** (*str*) – account type of azure, only support "China" and "Global".
- **account_name** (*str*) – account name of the azure.
- **account_key** (*str*) – account key of the azure.
- **container_name** (*str*) – container name of the azure.

Returns The cloud client of this dataset.

Return type *tensorbay.client.cloud_storage.CloudClient*

create_local_storage_config(*name*, *file_path*, *endpoint*)
Create a local auth storage config.

Parameters

- **name** (*str*) – The required local storage config name.
- **file_path** (*str*) – The dataset storage path under the local storage.
- **endpoint** (*str*) – The external IP address of the local storage.

Return type None

get_cloud_client(*name*)

Get a cloud client used for interacting with cloud platform.

Parameters **name** (*str*) – The required auth storage config name.

Returns The cloud client of this dataset.

Return type *tensorbay.client.cloud_storage.CloudClient*

create_dataset(*name: str, is_fusion: typing_extensions.Literal[False] = False, *, config_name: Optional[str] = 'None', alias: str = "''", is_public: bool = 'DEFAULT_IS_PUBLIC'*) → *tensorbay.client.dataset.DatasetClient*

create_dataset(*name: str, is_fusion: typing_extensions.Literal[True], *, config_name: Optional[str] = 'None', alias: str = "''", is_public: bool = 'DEFAULT_IS_PUBLIC'*) → *tensorbay.client.dataset.FusionDatasetClient*

create_dataset(*name: str, is_fusion: bool = False, *, config_name: Optional[str] = 'None', alias: str = "''", is_public: bool = 'DEFAULT_IS_PUBLIC'*) → 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.
- **config_name** – The auth storage config name.
- **alias** – Alias of the dataset, default is "".
- **is_public** – Whether the dataset is a public dataset.

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

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

list_dataset_names()

List names of all TensorBay datasets.

Returns The PagingList of all TensorBay dataset names.

Return type *tensorbay.client.lazy.PagingList*[str]

update_dataset(*name*, *, *alias=None*, *is_public=None*)

Update a TensorBay Dataset.

Parameters

- **name** (*str*) – Name of the dataset, unique for a user.
- **alias** (*Optional[str]*) – New alias of the dataset.
- **is_public** (*Optional[bool]*) – Whether the dataset is public.

Return type None

rename_dataset(*name*, *new_name*)

Rename a TensorBay Dataset with given name.

Parameters

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

Return type None

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

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

upload_dataset(*dataset: Union[tensorbay.dataset.dataset.Dataset, tensorbay.dataset.dataset.FusionDataset]*, *draft_number: Optional[int] = None*, *, *branch_name: Optional[str] = 'None'*, *jobs: int = '1'*, *skip_uploaded_files: bool = 'False'*, *quiet: 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.
- **draft_number** – The draft number.
- **branch_name** – The branch name.
- **jobs** – The number of the max workers in multi-thread upload.
- **skip_uploaded_files** – Set it to True to skip the uploaded files.
- **quiet** – Set to True to stop showing the upload process bar.

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

Raises

- **ValueError** – When uploading the dataset based on both draft number and branch name is not allowed.
- **Exception** – When Exception was raised during uploading dataset.

delete_dataset(*name*)

Delete a TensorBay dataset with given name.

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

Return type None

tensorbay.client.lazy

Lazy evaluation related classes.

class tensorbay.client.lazy.**LazyItem**(*page*, *data*)

Bases: Generic[tensorbay.client.lazy._T]

In paging lazy evaluation system, a LazyItem instance represents an element in a pagination.

If user wants to access the elememt, LazyItem will trigger the paging request to pull a page of elements and return the required element. All the pulled elements will be stored in different LazyItem instances and will not be requested again.

Parameters **page** – The page the item belongs to.

page

The parent *LazyPage* of this item.

data

The actual element stored in this item.

classmethod from_page(*page*)

Create a LazyItem instance from page.

Parameters **page** (tensorbay.client.lazy.LazyPage*[tensorbay.client.lazy._T]*) – The page of the element.

Returns The LazyItem instance which stores the input page.

Return type *tensorbay.client.lazy.LazyItem*[tensorbay.client.lazy._T]

classmethod from_data(*data*)

Create a LazyItem instance from data.

Parameters **data** (*tensorbay.client.lazy._T*) – The actual data needs to be stored in LazyItem.

Returns The LazyItem instance which stores the input data.

Return type *tensorbay.client.lazy.LazyItem*[tensorbay.client.lazy._T]

get()

Access the actual element represented by LazyItem.

If the element is already pulled from web, it will be return directly, otherwise this function will request for a page of elements to get the required elememt.

Returns The actual element this LazyItem instance represents.

Return type tensorbay.client.lazy._T

class tensorbay.client.lazy.**ReturnGenerator**(*generator*)
Bases: Generic[tensorbay.client.lazy._T, tensorbay.client.lazy._R]

ReturnGenerator is a generator wrap to get the return value easily.

Parameters **generator** – The generator needs to be wrapped.

value
The return value of the input generator.

Type tensorbay.client.lazy._R

class tensorbay.client.lazy.**LazyPage**(*offset*, *limit*, *func*)
Bases: Generic[tensorbay.client.lazy._T]

In paging lazy evaluation system, a LazyPage instance represents a page with elements.

LazyPage is used for sending paging request to pull a page of elements and storing them in different *LazyItem* instances.

Parameters

- **offset** – The offset of the page.
- **limit** – The limit of the page.
- **func** – A paging generator function, which takes offset<int> and limit<int> as inputs and returns a generator. The returned generator should yield the element user needs, and return the total count of the elements in the paging request.

items
The *LazyItem* list which represents a page of elements.

classmethod from_items(*offset*, *limit*, *func*, *item_contents*)
Create a LazyPage instance with the given items and generator function.

Parameters

- **offset** (*int*) – The offset of the page.
- **limit** (*int*) – The limit of the page.
- **func** (*Callable[[int, int], Generator[tensorbay.client.lazy._T, None, int]]*) – A paging generator function, which takes offset<int> and limit<int> as inputs and returns a generator. The returned generator should yield the element user needs, and return the total count of the elements in the paging request.
- **item_contents** (*Iterable[tensorbay.client.lazy._T]*) – The lazy item contents that need to be stored on this page.

Returns The LazyPage instance which stores the input items and function.

Return type *tensorbay.client.lazy.LazyPage*[tensorbay.client.lazy._T]

pull()
Send paging request to pull a page of elements and store them in *LazyItem*.

Return type None

class tensorbay.client.lazy.**InitPage**(*offset*, *limit*, *func*)
Bases: *tensorbay.client.lazy.LazyPage*[tensorbay.client.lazy._T]

In paging lazy evaluation system, InitPage is the page for initializing *PagingList*.

InitPage will send a paging request to pull a page of elements and storing them in different *LazyItem* instances when construction. And the totalCount of the page will also be stored in the instance.

Parameters

- **offset** – The offset of the page.
- **limit** – The limit of the page.
- **func** – A paging generator function, which takes offset<int> and limit<int> as inputs and returns a generator. The returned generator should yield the element user needs, and return the total count of the elements in the paging request.

items
The *LazyItem* list which represents a page of elements.

total_count
The totalCount of the paging request.

class tensorbay.client.lazy.**PagingList**(*func*, *limit*)
Bases: MutableSequence[tensorbay.client.lazy._T], *tensorbay.utility.repr.ReprMixin*

PagingList is a wrap of web paging request.

It follows the python MutableSequence protocal, which means it can be used like a python builtin list. And it provides features like lazy evaluation and cache.

Parameters

- **func** – A paging generator function, which takes offset<int> and limit<int> as inputs and returns a generator. The returned generator should yield the element user needs, and return the total count of the elements in the paging request.
- **limit** – The page size of each paging request.

insert(*index*, *value*)
Insert object before index.

Parameters

- **index** (*int*) – Position of the PagingList.
- **value** (*tensorbay.client.lazy._T*) – Element to be inserted into the PagingList.

Return type None

append(*value*)
Append object to the end of the PagingList.

Parameters value (*tensorbay.client.lazy._T*) – Element to be appended to the PagingList.

Return type None

reverse()
Reverse the items of the PagingList in place.

Return type None

pop(*index=- 1*)
Return the item at index (default last) and remove it from the PagingList.

Parameters index (*int*) – Position of the PagingList.

Returns Element to be removed from the PagingList.

Return type tensorbay.client.lazy._T

index(*value*, *start=0*, *stop=None*)
Return the first index of the value.

Parameters

- **value** (*Any*) – The value to be found.
- **start** (*int*) – The start index of the subsequence.
- **stop** (*Optional[int]*) – The end index of the subsequence.

Raises **ValueError** – When the value is not in the PagingList

Returns The first index of the value.

Return type int

count(*value*)

Return the number of occurrences of value.

Parameters **value** (*Any*) – The value needs to be counted.

Returns The number of occurrences of value.

Return type int

extend(*values*)

Extend PagingList by appending elements from the iterable.

Parameters **values** (*Iterable[tensorbay.client.lazy._T]*) – Elements to be extended into the PagingList.

Return type None

tensorbay.client.log

Logging utility functions.

Dump_request_and_response dumps http request and response.

class tensorbay.client.log.**RequestLogging**(*request*)

Bases: object

This class used to lazy load request to logging.

Parameters **request** (*requests.models.PreparedRequest*) – The request of the request.

Return type None

class tensorbay.client.log.**ResponseLogging**(*response*)

Bases: object

This class used to lazy load response to logging.

Parameters **response** (*requests.models.Response*) – The response of the request.

Return type None

tensorbay.client.log.**dump_request_and_response**(*response*)

Dumps http request and response.

Parameters **response** (*requests.models.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
"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": {}
}
"cost_time": 0.0813691616058
====================================================
```

Return type str

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*, *url=''*)

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

Return type None

static do(*method*, *url*, ***kwargs*)

Send a request.

Parameters

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

Returns Response of the request.

Return type requests.models.Response

open_api_do(*method*, *section*, *dataset_id=''*, ***kwargs*)

Send a request to the TensorBay Open API.

Parameters

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

Raises *ResponseError* – When the status code OpenAPI returns is unexpected.

Returns Response of the request.

Return type requests.models.Response

tensorbay.client.requests.**multithread_upload**(*function*, *arguments*, ***, *callback=None*, *jobs=1*, *pbar*)

Multi-thread upload framework.

Parameters

- **function** (*Callable[[tensorbay.client.requests._T], Optional[tensorbay.client.requests._R]]*) – The upload function.
- **arguments** (*Iterable[tensorbay.client.requests._T]*) – The arguments of the upload function.
- **callback** (*Optional[Callable[[Tuple[tensorbay.client.requests._R, ...]], None]]*) – The callback function.
- **jobs** (*int*) – The number of the max workers in multi-thread uploading procession.
- **pbar** (*tensorbay.utility.requests.Tqdm*) – The Tqdm instance for showing the upload process bar.

Return type None

class tensorbay.client.requests.**MultiCallbackTask**(**, function, callback, size=50*)

Bases: Generic[tensorbay.client.requests._T, tensorbay.client.requests._R]

A class for callbacking in multi-thread work.

Parameters

- **function** – The function of a single thread.
- **callback** – The callback function.
- **size** – The size of the task queue to send a callback.

work(*argument*)

Do the work of a single thread.

Parameters **argument** (*tensorbay.client.requests._T*) – The argument of the function.

Return type None

last_callback()

Send the last callback when all works have been done.

Return type None

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.**SegmentClientBase**(*name, dataset_client*)

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** (*str*) – Segment name.
- **dataset_client** (*Union[DatasetClient, FusionDatasetClient]*) – The dataset client.

Return type None

name
Segment name.

status
The status of the dataset client.

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

Parameters

- **name** (*str*) –
- **data_client** (DatasetClient) –

Return type None

upload_file(*local_path*, *target_remote_path=''*)
Upload data with local path to the draft.

Parameters

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

Return type None

upload_label(*data*)
Upload label with Data object to the draft.

Parameters **data** (tensorbay.dataset.data.Data) – The data object which represents the local file to upload.

Return type None

upload_data(*data*)
Upload Data object to the draft.

Parameters **data** (tensorbay.dataset.data.Data) – The *Data*.

Return type None

import_auth_data(*data*)
Import AuthData object to the draft.

Parameters **data** (tensorbay.dataset.data.AuthData) – The *Data*.

Return type None

copy_data(*source_remote_paths*, *target_remote_paths=None*, ***, *source_client=None*, *strategy='abort'*)
Copy data to this segment.

Parameters

- **source_remote_paths** (*Union[str, Iterable[str]]*) – The source remote paths of the copied data.
- **target_remote_paths** (*Union[None, str, Iterable[str]]*) – The target remote paths of the copied data. This argument is used to specify new remote paths of the copied data. If None, the remote path of the copied data will not be changed after copy.
- **source_client** (*Optional[tensorbay.client.segment.SegmentClient]*) – The source segment client of the copied data. This argument is used to specifies where the copied data comes from when the copied data is from another commit, draft, segment or even another dataset. If None, the copied data comes from this segment.
- **strategy** (*str*) – The strategy of handling the name conflict. There are three options:
  1. "abort": stop copying and raise exception;
  2. "override": the source data will override the origin data;
  3. "skip": keep the origin data.

Raises

- *InvalidParamsError* – When strategy is invalid.
- **ValueError** – When the type of target_remote_paths is not equal with source_remote_paths.

Return type None

move_data(*source_remote_paths*, *target_remote_paths=None*, ***, *source_client=None*, *strategy='abort'*)

Move data to this segment, also used to rename data.

Parameters

- **source_remote_paths** (*Union[str, Iterable[str]]*) – The source remote paths of the moved data.
- **target_remote_paths** (*Union[None, str, Iterable[str]]*) – The target remote paths of the moved data. This argument is used to specify new remote paths of the moved data. If None, the remote path of the moved data will not be changed after copy.
- **source_client** (*Optional[tensorbay.client.segment.SegmentClient]*) – The source segment client of the moved data. This argument is used to specifies where the moved data comes from when the moved data is from another segment. If None, the moved data comes from this segment.
- **strategy** (*str*) – The strategy of handling the name conflict. There are three options:
  1. "abort": stop copying and raise exception;
  2. "override": the source data will override the origin data;
  3. "skip": keep the origin data.

Raises

- *InvalidParamsError* – When strategy is invalid.
- **ValueError** – When the type or the length of target_remote_paths is not equal with source_remote_paths. Or when the dataset_id and drafter_number of source_client is not equal with the current segment client.

Return type None

list_data_paths()

List required data path in a segment in a certain commit.

Returns The PagingList of data paths.

Return type *tensorbay.client.lazy.PagingList*[str]

get_data(*remote_path*)
Get required Data object from a dataset segment.

Parameters **remote_path** (*str*) – The remote paths of the required data.

Returns *RemoteData*.

Raises ***ResourceNotExistError*** – When the required data does not exist.

Return type *tensorbay.dataset.data.RemoteData*

list_data()
List required Data object in a dataset segment.

Returns The PagingList of *RemoteData*.

Return type *tensorbay.client.lazy.PagingList*[*tensorbay.dataset.data.RemoteData*]

delete_data(*remote_path*)
Delete data of a segment in a certain commit with the given remote paths.

Parameters **remote_path** (*str*) – The remote path of data in a segment.

Return type None

list_urls()
List the data urls in this segment.

Returns The PagingList of urls.

Return type *tensorbay.client.lazy.PagingList*[str]

list_mask_urls(*mask_type*)
List the mask urls in this segment.

Parameters **mask_type** (*str*) – The required mask type, the supported types are `SEMANTIC_MASK`, `INSTANCE_MASK` and `PANOPTIC_MASK`

Returns The PagingList of mask urls.

Return type *tensorbay.client.lazy.PagingList*[Optional[str]]

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

Parameters

- **name** (*str*) –
- **data_client** (FusionDatasetClient) –

Return type None

get_sensors()
Return the sensors in a fusion segment client.

Returns The `sensors` in the fusion segment client.

Return type *tensorbay.sensor.sensor.Sensors*

upload_sensor(*sensor*)
Upload sensor to the draft.

Parameters **sensor** (`tensorbay.sensor.sensor.Sensor`) – The sensor to upload.

Return type None

delete_sensor(*sensor_name*)
Delete a TensorBay sensor of the draft with the given sensor name.

Parameters **sensor_name** (*str*) – The TensorBay sensor to delete.

Return type None

upload_frame(*frame*, *timestamp=None*)
Upload frame to the draft.

Parameters

- **frame** (`tensorbay.dataset.frame.Frame`) – The *Frame* to upload.
- **timestamp** (*Optional[float]*) – The mark to sort frames, supporting timestamp and float.

Raises ***FrameError*** – When lacking frame id or frame id conflicts.

Return type None

list_frames()
List required frames in the segment in a certain commit.

Returns The PagingList of *Frame*.

Return type *tensorbay.client.lazy.PagingList*[*tensorbay.dataset.frame.Frame*]

delete_frame(*frame_id*)
Delete a frame of a segment in a certain commit with the given frame id.

Parameters **frame_id** (*Union[str, ulid.ulid.ULID]*) – The id of a frame in a segment.

Return type None

list_urls()
List the data urls in this segment.

Returns The PagingList of url dict, which key is the sensor name, value is the url.

Return type *tensorbay.client.lazy.PagingList*[Dict[str, str]]

tensorbay.client.status

Class Status.

class tensorbay.client.status.**Status**(*branch_name=None*, ***, *draft_number=None*, *commit_id=None*)
Bases: `object`

This class defines the basic concept of the status.

Parameters

- **branch_name** (*Optional[str]*) – The branch name.
- **draft_number** (*Optional[int]*) – The draft number (if the status is draft).

- **commit_id** (*Optional[str]*) – The commit ID (if the status is commit).

Return type None

property is_draft: bool
Return whether the status is draft, True for draft, False for commit.

Returns whether the status is draft, True for draft, False for commit.

property draft_number: Optional[int]
Return the draft number.

Returns The draft number.

property commit_id: Optional[str]
Return the commit ID.

Returns The commit ID.

get_status_info()
Get the dict containing the draft number or commit ID.

Returns A dict containing the draft number or commit ID.

Return type Dict[str, Any]

check_authority_for_commit()
Check whether the status is a legal commit.

Raises *StatusError* – When the status is not a legal commit.

Return type None

check_authority_for_draft()
Check whether the status is a legal draft.

Raises *StatusError* – When the status is not a legal draft.

Return type None

checkout(*commit_id=None*, *draft_number=None*)
Checkout to commit or draft.

Parameters

- **commit_id** (*Optional[str]*) – The commit ID.
- **draft_number** (*Optional[int]*) – The draft number.

Return type None

tensorbay.client.struct

User, Commit, Tag, Branch and Draft classes.

User defines the basic concept of a user with an action.

Commit defines the structure of a commit.

Tag defines the structure of a commit tag.

Branch defines the structure of a branch.

Draft defines the structure of a draft.

class tensorbay.client.struct.**TeamInfo**(*name*, ***, *email=None*, *description=''*)

Bases: *tensorbay.utility.name.NameMixin*

This class defines the basic concept of a TensorBay team.

Parameters

- **name** (*str*) – The name of the team.
- **email** (*Optional[str]*) – The email of the team.
- **description** (*str*) – The description of the team.

Return type None

classmethod loads(*contents*)

Loads a *TeamInfo* instance from the given contents.

Parameters **contents** (*Dict[str, Any]*) – A dict containing all the information of the commit:

```
{
    "name": <str>
    "email": <str>
    "description": <str>
}
```

Returns A *TeamInfo* instance containing all the information in the given contents.

Return type tensorbay.client.struct._T

dumps()

Dumps all the information into a dict.

Returns

A dict containing all the information of the team:

```
{
        "name": <str>
        "email": <str>
        "description": <str>
}
```

Return type Dict[str, Any]

class tensorbay.client.struct.**UserInfo**(*name*, ***, *email=None*, *mobile=None*, *description=''*, *team=None*)

Bases: *tensorbay.utility.name.NameMixin*

This class defines the basic concept of a TensorBay user.

Parameters

- **name** (*str*) – The nickname of the user.
- **email** (*Optional[str]*) – The email of the user.
- **mobile** (*Optional[str]*) – The mobile of the user.
- **description** (*str*) – The description of the user.
- **team** (*Optional[tensorbay.client.struct.TeamInfo]*) – The team of the user.

Return type None

classmethod loads(*contents*)

Loads a *UserInfo* instance from the given contents.

Parameters **contents** (*Dict[str, Any]*) – A dict containing all the information of the commit:

```
{
    "name": <str>
    "email": <str>
    "mobile": <str>
    "description": <str>
    "team": {  <dict>
        "name": <str>
        "email": <str>
        "description": <str>
    }
}
```

Returns A *UserInfo* instance containing all the information in the given contents.

Return type tensorbay.client.struct._T

dumps()

Dumps all the information into a dict.

Returns

A dict containing all the information of the user:

```
{
        "name": <str>
        "email": <str>
        "mobile": <str>
        "description": <str>
        "team": {  <dict>
            "name": <str>
            "email": <str>
            "description": <str>
        }
}
```

Return type Dict[str, Any]

class tensorbay.client.struct.**User**(*name*, *date*)

Bases: *tensorbay.utility.attr.AttrsMixin*, *tensorbay.utility.repr.ReprMixin*

This class defines the basic concept of a user with an action.

Parameters

- **name** (*str*) – The name of the user.
- **date** (*int*) – The date of the user action.

Return type None

classmethod loads(*contents*)

Loads a *User* instance from the given contents.

Parameters **contents** (*Dict[str, Any]*) – A dict containing all the information of the commit:

```
{
    "name": <str>
    "date": <int>
}
```

Returns A *User* instance containing all the information in the given contents.

Return type tensorbay.client.struct._T

dumps()

Dumps all the user information into a dict.

Returns

A dict containing all the information of the user:

```
{
    "name": <str>
    "date": <int>
}
```

Return type Dict[str, Any]

class tensorbay.client.struct.**Commit**(*commit_id*, *parent_commit_id*, *title*, *description*, *committer*)

Bases: *tensorbay.utility.attr.AttrsMixin*, *tensorbay.utility.repr.ReprMixin*

This class defines the structure of a commit.

Parameters

- **commit_id** (*str*) – The commit id.
- **parent_commit_id** (*Optional[str]*) – The parent commit id.
- **title** (*str*) – The commit title.
- **description** (*str*) – The commit description.
- **committer** (tensorbay.client.struct.User) – The commit user.

Return type None

classmethod loads(*contents*)

Loads a *Commit* instance for the given contents.

Parameters contents (*Dict[str, Any]*) – A dict containing all the information of the commit:

```
{
    "commitId": <str>
    "parentCommitId": <str> or None
    "title": <str>
    "description": <str>
    "committer": {
        "name": <str>
        "date": <int>
    }
}
```

Returns A *Commit* instance containing all the information in the given contents.

Return type tensorbay.client.struct._T

dumps()

Dumps all the commit information into a dict.

Returns

A dict containing all the information of the commit:

```
{
    "commitId": <str>
    "parentCommitId": <str> or None
    "title": <str>
    "description": <str>
    "committer": {
        "name": <str>
        "date": <int>
    }
}
```

Return type Dict[str, Any]

class tensorbay.client.struct.Tag(*name*, *commit_id*, *parent_commit_id*, *title*, *description*, *committer*)

Bases: `tensorbay.client.struct._NamedCommit`

This class defines the structure of the tag of a commit.

Parameters

- **name** (*str*) – The name of the tag.
- **commit_id** (*str*) – The commit id.
- **parent_commit_id** (*Optional[str]*) – The parent commit id.
- **title** (*str*) – The commit title.
- **description** (*str*) – The commit description.
- **committer** (`tensorbay.client.struct.User`) – The commit user.

Return type None

class tensorbay.client.struct.Branch(*name*, *commit_id*, *parent_commit_id*, *title*, *description*, *committer*)

Bases: `tensorbay.client.struct._NamedCommit`

This class defines the structure of a branch.

Parameters

- **name** (*str*) – The name of the branch.
- **commit_id** (*str*) – The commit id.
- **parent_commit_id** (*Optional[str]*) – The parent commit id.
- **title** (*str*) – The commit title.
- **description** (*str*) – The commit description.
- **committer** (`tensorbay.client.struct.User`) – The commit user.

Return type None

class tensorbay.client.struct.Draft(*number*, *title*, *branch_name*, *status*, *parent_commit_id*, *author*, *updated_at*, *description=''*)

Bases: *tensorbay.utility.attr.AttrsMixin*, *tensorbay.utility.repr.ReprMixin*

This class defines the basic structure of a draft.

Parameters

- **number** (*int*) – The number of the draft.
- **title** (*str*) – The title of the draft.
- **branch_name** (*str*) – The branch name.
- **status** (*str*) – The status of the draft.
- **parent_commit_id** (*str*) – The parent commit id.
- **author** (tensorbay.client.struct.User) – The author of the draft.
- **updated_at** (*int*) – The time of last update.
- **description** (*str*) – The draft description.

Return type None

classmethod loads(*contents*)

Loads a *Draft* instance from the given contents.

Parameters contents (*Dict[str, Any]*) – A dict containing all the information of the draft:

```
{
    "number": <int>
    "title": <str>
    "branchName": <str>
    "status": "OPEN", "CLOSED" or "COMMITTED"
    "parentCommitId": <str>
    "author": {
        "name": <str>
        "date": <int>
    }
    "updatedAt": <int>
    "description": <str>
}
```

Returns A *Draft* instance containing all the information in the given contents.

Return type tensorbay.client.struct._T

dumps()

Dumps all the information of the draft into a dict.

Returns

A dict containing all the information of the draft:

```
{
    "number": <int>
    "title": <str>
    "branchName": <str>
    "status": "OPEN", "CLOSED" or "COMMITTED"
    "parentCommitId": <str>
    "author": {
        "name": <str>
        "date": <int>
```

(continues on next page)

(continued from previous page)

```
    }
    "updatedAt": <int>
    "description": <str>
}
```

Return type Dict[str, Any]

tensorbay.client.version

TensorBay dataset version control related classes.

class tensorbay.client.version.**VersionControlMixin**

Bases: `object`

A mixin class supporting version control methods.

Parameters

- **_dataset_id** – Dataset ID.
- **_client** – The client to interact between local and TensorBay.
- **_status** – The version control status of the dataset.

checkout(*revision=None*, *draft_number=None*)

Checkout to commit or draft.

Parameters

- **revision** (*Optional[str]*) – The information to locate the specific commit, which can be the commit id, the branch, or the tag.
- **draft_number** (*Optional[int]*) – The draft number.

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

Return type None

commit(*title*, *description=''*, ***, *tag=None*)

Commit the draft.

Commit the draft based on the draft number stored in the dataset client. Then the dataset client will change the status to "commit" and store the branch name and commit id.

Parameters

- **title** (*str*) – The commit title.
- **description** (*str*) – The commit description.
- **tag** (*Optional[str]*) – A tag for current commit.

Return type None

create_draft(*title*, *description=''*, *branch_name=None*)

Create a draft.

Create a draft with the branch name. If the branch name is not given, create a draft based on the branch name stored in the dataset client. Then the dataset client will change the status to "draft" and store the branch name and draft number.

Parameters

- **title** (*str*) – The draft title.
- **description** (*str*) – The draft description.
- **branch_name** (*Optional[str]*) – The branch name.

Returns The draft number of the created draft.

Raises *StatusError* – When creating the draft without basing on a branch.

Return type int

get_draft(*draft_number=None*)

Get the certain draft with the given draft number.

Get the certain draft with the given draft number. If the draft number is not given, get the draft based on the draft number stored in the dataset client.

Parameters **draft_number** (*Optional[int]*) – The required draft number. If is not given, get the current draft.

Returns The *Draft* instance with the given number.

Raises

- **TypeError** – When the given draft number is illegal.
- *ResourceNotExistError* – When the required draft does not exist.

Return type *tensorbay.client.struct.Draft*

list_drafts(*status='OPEN'*, *branch_name=None*)

List all the drafts.

Parameters

- **status** (*Optional[str]*) – The draft status which includes "OPEN", "CLOSED", "COMMITTED", "ALL" and None. where None means listing open drafts.
- **branch_name** (*Optional[str]*) – The branch name.

Returns The PagingList of *drafts*.

Return type *tensorbay.client.lazy.PagingList*[*tensorbay.client.struct.Draft*]

update_draft(*draft_number=None*, ***, *title=None*, *description=None*)

Update the draft.

Parameters

- **draft_number** (*Optional[int]*) – The updated draft number. If is not given, update the current draft.
- **title** (*Optional[str]*) – The title of the draft.
- **description** (*Optional[str]*) – The description of the draft.

Return type None

close_draft(*number*)

Close the draft.

Parameters **number** (*int*) – The draft number.

Raises *StatusError* – When closing the current draft.

Return type None

squash_and_merge(*title*, *description=''*, *, *source_branch_name*, *target_branch_name=None*, *strategy='abort'*)

Squash and merge.

Squash commits in source branch, then merge into target branch by creating a new draft. If the target branch name is not given, the draft will be based on the branch name stored in the dataset client. And during merging, the conflicts between branches can be resolved in three different strategies: "abort", "override" and "skip".

Parameters

- **title** (*str*) – The draft title.
- **description** (*str*) – The draft description.
- **source_branch_name** (*str*) – The name of the branch to be squashed.
- **target_branch_name** (*Optional[str]*) – The target branch name of the merge operation.
- **strategy** (*Optional[str]*) – The strategy of handling the branch conflict. There are three options:

1. "abort": abort the opetation;
2. "override": the squashed branch will override the target branch;
3. "skip": keep the origin branch.

Raises *StatusError* – When squashing and merging without basing on a branch.

Returns The draft number of the new draft.

Return type int

get_commit(*revision=None*)

Get the certain commit with the given revision.

Get the certain commit with the given revision. If the revision is not given, get the commit based on the commit id stored in the dataset client.

Parameters **revision** (*Optional[str]*) – The information to locate the specific commit, which can be the commit id, the branch name, or the tag name. If is not given, get the current commit.

Returns The *Commit* instance with the given revision.

Raises

- **TypeError** – When the given revision is illegal.
- *ResourceNotExistError* – When the required commit does not exist.

Return type *tensorbay.client.struct.Commit*

list_commits(*revision=None*)

List the commits.

Parameters **revision** (*Optional[str]*) – The information to locate the specific commit, which can be the commit id, the branch name, or the tag name. If is given, list the commits before the given commit. If is not given, list the commits before the current commit.

Raises **TypeError** – When the given revision is illegal.

Returns The PagingList of *commits*.

Return type *tensorbay.client.lazy.PagingList*[*tensorbay.client.struct.Commit*]

`create_branch`(*name*, *revision=None*)
Create a branch.

Create a branch based on a commit with the given revision. If the revision is not given, create a branch based on the commit id stored in the dataset client. Then the dataset client will change the status to "commit" and store the branch name and the commit id.

Parameters

- **name** (*str*) – The branch name.
- **revision** (*Optional[str]*) – The information to locate the specific commit, which can be the commit id, the branch name, or the tag name. If the revision is not given, create the branch based on the current commit.

Return type None

`get_branch`(*name*)
Get the branch with the given name.

Parameters **name** (*str*) – The required branch name.

Returns The `Branch` instance with the given name.

Raises

- **TypeError** – When the given branch is illegal.
- ***ResourceNotExistError*** – When the required branch does not exist.

Return type *tensorbay.client.struct.Branch*

`list_branches`()
List the information of branches.

Returns The PagingList of `branches`.

Return type *tensorbay.client.lazy.PagingList*[*tensorbay.client.struct.Branch*]

`delete_branch`(*name*)
Delete a branch.

Delete the branch with the given branch name. Note that deleting the branch with the name which is stored in the current dataset client is not allowed.

Parameters **name** (*str*) – The name of the branch to be deleted.

Raises ***StatusError*** – When deleting the current branch.

Return type None

`create_tag`(*name*, *revision=None*)
Create a tag for a commit.

Create a tag for a commit with the given revision. If the revision is not given, create a tag based on the commit id stored in the dataset client.

Parameters

- **name** (*str*) – The tag name to be created for the specific commit.
- **revision** (*Optional[str]*) – The information to locate the specific commit, which can be the commit id, the branch name, or the tag name. If the revision is not given, create the tag for the current commit.

Return type None

get_tag(*name*)
Get the certain tag with the given name.

Parameters **name** (*str*) – The required tag name.

Returns The *Tag* instance with the given name.

Raises

- **TypeError** – When the given tag is illegal.
- ***ResourceNotExistError*** – When the required tag does not exist.

Return type *tensorbay.client.struct.Tag*

list_tags()
List the information of tags.

Returns The PagingList of *tags*.

Return type *tensorbay.client.lazy.PagingList*[*tensorbay.client.struct.Tag*]

delete_tag(*name*)
Delete a tag.

Parameters **name** (*str*) – The tag name to be deleted for the specific commit.

Return type None

tensorbay.client.diff

Class about the diff.

DiffBase defines the basic structure of a diff.

NotesDiff defines the basic structure of a brief diff of notes.

CatalogDiff defines the basic structure of a brief diff of catalog.

FileDiff defines the basic structure of a brief diff of data file.

LabelDiff defines the basic structure of a brief diff of data label.

SensorDiff defines the basic structure of a brief diff of sensor.

DataDiff defines the basic structure of a brief diff of data.

SegmentDiff defines the basic structure of a brief diff of a segment.

DatasetDiff defines the basic structure of a brief diff of a dataset.

class tensorbay.client.diff.**DiffBase**(*action*)
Bases: *tensorbay.utility.attr.AttrsMixin*, *tensorbay.utility.repr.ReprMixin*

This class defines the basic structure of a diff.

Parameters **action** (*str*) –

Return type None

action
The concrete action.

Type str

classmethod loads(*contents*)
Loads a *DiffBase* instance from the given contents.

Parameters **contents** (*Dict[str, Any]*) – A dict containing all the information of the diff:

```
{
    "action": <str>
}
```

Returns A *DiffBase* instance containing all the information in the given contents.

Return type tensorbay.client.diff._T

dumps()

Dumps all the information of the diff into a dict.

Returns

A dict containing all the information of the diff:

```
{
    "action": <str>
}
```

Return type Dict[str, Any]

class tensorbay.client.diff.**NotesDiff**(*action*)

Bases: *tensorbay.client.diff.DiffBase*

This class defines the basic structure of a brief diff of notes.

Parameters **action** (*str*) –

Return type None

class tensorbay.client.diff.**CatalogDiff**(*action*)

Bases: *tensorbay.client.diff.DiffBase*

This class defines the basic structure of a brief diff of catalog.

Parameters **action** (*str*) –

Return type None

class tensorbay.client.diff.**FileDiff**(*action*)

Bases: *tensorbay.client.diff.DiffBase*

This class defines the basic structure of a brief diff of data file.

Parameters **action** (*str*) –

Return type None

class tensorbay.client.diff.**LabelDiff**(*action*)

Bases: *tensorbay.client.diff.DiffBase*

This class defines the basic structure of a brief diff of data label.

Parameters **action** (*str*) –

Return type None

class tensorbay.client.diff.**SensorDiff**(*action*)

Bases: *tensorbay.client.diff.DiffBase*

This class defines the basic structure of a brief diff of sensor.

Parameters **action** (*str*) –

Return type None

`class tensorbay.client.diff.DataDiff(action)`

Bases: *tensorbay.client.diff.DiffBase*

This class defines the basic structure of a diff statistic.

Parameters **action** (*str*) –

Return type None

remote_path

The remote path.

Type str

action

The action of data.

Type str

file

The brief diff information of the file.

Type *tensorbay.client.diff.FileDiff*

label

The brief diff information of the labels.

Type *tensorbay.client.diff.LabelDiff*

`classmethod loads(contents)`

Loads a *DataDiff* instance from the given contents.

Parameters **contents** (*Dict[str, Any]*) – A dict containing all the brief diff information of data:

```
{
    "remotePath": <str>,
    "action": <str>,
    "file": {
        "action": <str>
    },
    "label": {
        "action": <str>
    }
}
```

Returns A *DataDiff* instance containing all the information in the given contents.

Return type tensorbay.client.diff._T

`dumps()`

Dumps all the brief diff information of data into a dict.

Returns

A dict containing all the brief diff information of data:

```
{
    "remotePath": <str>,
    "action": <str>,
```

(continues on next page)

(continued from previous page)

```
    "file": {
        "action": <str>
    },
    "label": {
        "action": <str>
    }
}
```

Return type Dict[str, Any]

class tensorbay.client.diff.**SegmentDiff**(*name*, *action*, *data*)

Bases: *tensorbay.utility.user.UserSequence*[*tensorbay.client.diff.DataDiff*], *tensorbay.utility.name.NameMixin*

This class defines the basic structure of a brief diff of a segment.

Parameters

- **name** – The segment name.
- **action** – The action of a segment.

class tensorbay.client.diff.**DatasetDiff**(*name*, *segments*)

Bases: Sequence[*tensorbay.client.diff.SegmentDiff*], *tensorbay.utility.name.NameMixin*

This class defines the basic structure of a brief diff of a dataset.

Parameters

- **name** – The segment name.
- **action** – The action of a segment.

tensorbay.client.profile

Statistical.

Profile is a class used to save statistical summary.

tensorbay.client.profile.**format_size**(*size*)

Format a byte count as a human readable file size.

Parameters **size** (*float*) – The size to format in bytes.

Returns The human readable string.

Return type str

class tensorbay.client.profile.**Profile**

Bases: object

This is a class used to save statistical summary.

Return type None

save(*path*, *file_type='txt'*)

Save the statistical summary into a file.

Parameters

- **path** (*str*) – The file local path.
- **file_type** (*str*) – Type of the save file, only support 'txt', 'json', 'csv'.

Return type None

start(*multiprocess=False*)
Start statistical record.

Parameters **multiprocess** (*bool*) – Whether the records is in a multi-process environment.

Return type None

stop()
Stop statistical record.

Return type None

tensorbay.client.statistics

Class Statistics.

Statistics defines the basic structure of the label statistics obtained by `DatasetClientBase.get_label_statistics()`.

class tensorbay.client.statistics.**Statistics**(*data*)
Bases: *tensorbay.utility.user.UserMapping*[str, Any]

This class defines the basic structure of the label statistics.

Parameters **data** – The dict containing label statistics.

dumps()
Dumps the label statistics into a dict.

Returns A dict containing all the information of the label statistics.

Return type Dict[str, Any]

Examples

```
>>> label_statistics = Statistics(
...     {
...         'BOX3D': {
...             'quantity': 1234
...         },
...         'KEYPOINTS2D': {
...             'quantity': 43234,
...             'categories': [
...                 {
...                     'name': 'person.person',
...                     'quantity': 43234
...                 }
...             ]
...         }
...     }
... )
>>> label_statistics.dumps()
... {
...     'BOX3D': {
...         'quantity': 1234
```

(continues on next page)

(continued from previous page)

```
...     },
...     'KEYPOINTS2D': {
...         'quantity': 43234,
...         'categories': [
...             {
...                 'name': 'person.person',
...                 'quantity': 43234
...             }
...         ]
...     }
... }
```

1.24.2 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.**DataBase**(*timestamp=None*)

Bases: `tensorbay.utility.repr.ReprMixin`

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

Parameters **timestamp** (*Optional[float]*) – The timestamp for the file.

Return type None

timestamp

The timestamp for the file.

label

The `Label` instance that contains all the label information of the file.

class tensorbay.dataset.data.**Data**(*local_path*, ***, *target_remote_path=None*, *timestamp=None*)

Bases: `tensorbay.dataset.data.DataBase`, `tensorbay.utility.file.FileMixin`

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** (*str*) – The file local path.
- **target_remote_path** (*Optional[str]*) – The file remote path after uploading to tensorbay.
- **timestamp** (*Optional[float]*) – The timestamp for the file.

Return type None

path

The file local path.

timestamp
The timestamp for the file.

label
The *Label* instance that contains all the label information of the file.

target_remote_path
The target remote path of the data.

get_callback_body()
Get the callback request body for uploading.

Returns

The callback request body, which look like:

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

Return type Dict[str, Any]

class tensorbay.dataset.data.**RemoteData**(*remote_path*, ***, *timestamp=None*, *url=None*, *cache_path=''*)
Bases: *tensorbay.dataset.data.DataBase*, *tensorbay.utility.file.RemoteFileMixin*

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** (*str*) – The file remote path.
- **timestamp** (*Optional[float]*) – The timestamp for the file.
- **url** (*Optional[tensorbay.utility.file.URL]*) – The URL instance used to get and update url.
- **cache_path** (*str*) – The path to store the cache.

Return type None

path
The file remote path.

timestamp
The timestamp for the file.

label
The *Label* instance that contains all the label information of the file.

url
The *URL* instance used to get and update url.

classmethod from_response_body(*body*, ***, *url=None*, *cache_path=''*)
Loads a *RemoteData* object from a response body.

Parameters

- **body** (*Dict[str, Any]*) – The response body which contains the information of a remote data, whose format should be like:

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

- **url** (*Optional[tensorbay.utility.file.URL]*) – The URL instance used to get and update url.
- **cache_path** (*str*) – The path to store the cache.

Returns The loaded *RemoteData* object.

Return type tensorbay.dataset.data._T

class tensorbay.dataset.data.**AuthData**(*cloud_path*, ***, *target_remote_path=None*, *timestamp=None*, *url=None*)
Bases: *tensorbay.dataset.data.DataBase*, *tensorbay.utility.file.RemoteFileMixin*

AuthData is a combination of a specific cloud storaged file and its label.

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

An AuthData instance contains one or several types of labels.

Parameters

- **cloud_path** (*str*) – The cloud file path.
- **target_remote_path** (*Optional[str]*) – The file remote path after uploading to tensorbay.
- **timestamp** (*Optional[float]*) – The timestamp for the file.
- **url** (*Optional[tensorbay.utility.file.URL]*) – The URL instance used to get and update url.

Return type None

path
The cloud file path.

timestamp
The timestamp for the file.

label
The *Label* instance that contains all the label information of the file.

url
The *URL* instance used to get and update url.

get_callback_body()
Get the callback request body for uploading.

Returns

The callback request body, which looks like:

```
{
    "cloudPath": <str>,
    "remotePath": <str>,
    "label": {
        "CLASSIFICATION": {...},
        "BOX2D": {...},
        "BOX3D": {...},
        "POLYGON": {...},
        "POLYLINE2D": {...},
        "KEYPOINTS2D": {...},
        "SENTENCE": {...}
    }
}
```

Return type Dict[str, Any]

tensorbay.dataset.dataset

Notes, DatasetBase, Dataset and FusionDataset.

Notes contains the basic information of a *DatasetBase*.

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.**Notes**(*is_continuous=False*, *bin_point_cloud_fields=None*)
Bases: *tensorbay.utility.attr.AttrsMixin*, *tensorbay.utility.repr.ReprMixin*

This is a class stores the basic information of *DatasetBase*.

Parameters

- **is_continuous** (*bool*) – Whether the data inside the dataset is time-continuous.

- **bin_point_cloud_fields** (*Optional[List[str]]*) – The field names of the bin point cloud files in the dataset.

Return type None

classmethod loads(*contents*)

Loads a *Notes* instance from the given contents.

Parameters contents (*Dict[str, Any]*) – The given dict containing the dataset notes:

```
{
    "isContinuous":            <boolean>
    "binPointCloudFields": [   <array> or null
            <field_name>,      <str>
            ...
    ]
}
```

Returns The loaded *Notes* instance.

Return type tensorbay.dataset.dataset._T

keys()

Return the valid keys within the notes.

Returns The valid keys within the notes.

Return type KeysView[str]

dumps()

Dumps the notes into a dict.

Returns

A dict containing all the information of the Notes:

```
{
    "isContinuous":           <boolean>
    "binPointCloudFields": [  <array> or null
        <field_name>,         <str>
        ...
    ]
}
```

Return type Dict[str, Any]

class tensorbay.dataset.dataset.DatasetBase(*name*, *gas=None*, *revision=None*)

Bases: Sequence[tensorbay.dataset.dataset._T], *tensorbay.utility.name.NameMixin*

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.
- **gas** – The *GAS* client for getting a remote dataset.
- **revision** – The revision of the remote dataset.

catalog
The `Catalog` of the dataset.

notes
The `Notes` of the dataset.

property cache_enabled: bool
Whether the cache is enabled.

Returns Whether the cache is enabled.

enable_cache(*cache_path=''*)
Enable cache when open the remote data of the dataset.

Parameters **cache_path** (*str*) – The path to store the cache.

Return type None

keys()
Get all segment names.

Returns A tuple containing all segment names.

Return type Tuple[str, . . .]

load_catalog(*filepath*)
Load catalog from a json file.

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

Return type None

add_segment(*segment*)
Add a segment to the dataset.

Parameters **segment** (*tensorbay.dataset.dataset._T*) – The segment to be added.

Return type None

class tensorbay.dataset.dataset.**Dataset**(*name*, *gas=None*, *revision=None*)
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='default'*)
Create a segment with the given name.

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

Returns The created `Segment`.

Return type *tensorbay.dataset.segment.Segment*

class tensorbay.dataset.dataset.**FusionDataset**(*name*, *gas=None*, *revision=None*)
Bases: `tensorbay.dataset.dataset.DatasetBase`[`tensorbay.dataset.segment.FusionSegment`]

This class defines the concept of fusion dataset.

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

create_segment(*segment_name='default'*)
Create a fusion segment with the given name.

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

Returns The created `FusionSegment`.

Return type *tensorbay.dataset.segment.FusionSegment*

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 `Sensors`.

class tensorbay.dataset.segment.**Segment**(*name='default'*, *client=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=<function Segment.<lambda>>*, *reverse=False*)

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** (*Callable[[DataBase._Type], Any]*) – 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** (*bool*) – The reverse flag can be set as True to sort in descending order.

Raises **NotImplementedError** – The sort method for segment init from client is not supported yet.

Return type None

class tensorbay.dataset.segment.**FusionSegment**(*name='default'*, *client=None*)

Bases: *tensorbay.utility.name.NameMixin*, *tensorbay.utility.user.UserMutableSequence*[*tensorbay.dataset.frame.Frame*]

This class defines the concept of fusion segment.

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

Besides, a fusion segment contains multiple *Sensors* correspoinding to the *Data* under each *Frame*.

If the segment is inside of a time-continuous *FusionDataset*, the time continuity of the frames should be indicated by the index inside the fusion segment.

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

To initialize a *FusionSegment* and add a *Frame* to it:

```
fusion_segment = FusionSegment(fusion_segment_name)
frame = Frame()
...
fusion_segment.append(frame)
```

Parameters

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

property sensors: *tensorbay.sensor.sensor.Sensors*

Return the sensors of the fusion segment.

Returns The *Sensors* of the fusion dataset.

tensorbay.dataset.frame

Frame.

Frame is a concept in *FusionDataset*.

It is the structure that composes a *FusionSegment*, and consists of multiple *Data* collected at the same time from different sensors.

class tensorbay.dataset.frame.**Frame**(*frame_id=None*)

Bases: *tensorbay.utility.user.UserMutableMapping*[str, DataBase._Type]

This class defines the concept of frame.

Frame is a concept in *FusionDataset*.

It is the structure that composes *FusionSegment*, and consists of multiple *Data* collected at the same time corresponding to different sensors.

Since *Frame* extends *UserMutableMapping*, its basic operations are the same as a dictionary's.

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

```
frame = Frame()
frame[sensor_name] = Data()
```

classmethod from_response_body(*body*, *url_index*, *urls*, ***, *cache_path=''*)

Loads a *Frame* object from a response body.

Parameters

- **body** (*Dict[str, Any]*) – The response body which contains the information of a frame, whose format should be like:

```
{
    "frameId": <str>,
    "frame": [
        {
            "sensorName": <str>,
            "remotePath": <str>,
            "timestamp": <float>,
            "url": <str>,
            "label": {...}
        },
        ...
        ...
    ]
}
```

- **url_index** (*int*) – The index of the url.
- **urls** (tensorbay.client.lazy.LazyPage*[Dict[str, str]]*) – A sequence of mappings which key is the sensor name and value is the url.
- **cache_path** (*str*) – The path to store the cache.

Returns The loaded *Frame* object.

Return type tensorbay.dataset.frame._T

1.24.3 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*, *ymin*, *xmax*, *ymax*)

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.

Examples

```
>>> Box2D(1, 2, 3, 4)
Box2D(1, 2, 3, 4)
```

static iou(*box1*, *box2*)

Calculate the intersection over union of two 2D boxes.

Parameters

- **box1** (tensorbay.geometry.box.Box2D) – A 2D box.
- **box2** (tensorbay.geometry.box.Box2D) – A 2D box.

Returns The intersection over union between the two input boxes.

Return type float

Examples

```
>>> box2d_1 = Box2D(1, 2, 3, 4)
>>> box2d_2 = Box2D(2, 2, 3, 4)
>>> Box2D.iou(box2d_1, box2d_2)
0.5
```

classmethod from_xywh(*x*, *y*, *width*, *height*)

Create a *Box2D* instance from the top-left vertex and the width and the height.

Parameters

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

Returns The created *Box2D* instance.

Return type tensorbay.geometry.box._B2

Examples

```
>>> Box2D.from_xywh(1, 2, 3, 4)
Box2D(1, 2, 4, 6)
```

classmethod loads(*contents*)

Load a *Box2D* from a dict containing coordinates of the 2D box.

Parameters **contents** (*Mapping[str, float]*) – A dict containing coordinates of a 2D box.

Returns The loaded *Box2D* object.

Return type tensorbay.geometry.box._B2

Examples

```
>>> contents = {"xmin": 1.0, "ymin": 2.0, "xmax": 3.0, "ymax": 4.0}
>>> Box2D.loads(contents)
Box2D(1.0, 2.0, 3.0, 4.0)
```

property xmin: float

Return the minimum x coordinate.

Returns Minimum x coordinate.

Examples

```
>>> box2d = Box2D(1, 2, 3, 4)
>>> box2d.xmin
1
```

property ymin: float

Return the minimum y coordinate.

Returns Minimum y coordinate.

Examples

```
>>> box2d = Box2D(1, 2, 3, 4)
>>> box2d.ymin
2
```

property xmax: float

Return the maximum x coordinate.

Returns Maximum x coordinate.

Examples

```
>>> box2d = Box2D(1, 2, 3, 4)
>>> box2d.xmax
3
```

property ymax: float

Return the maximum y coordinate.

Returns Maximum y coordinate.

Examples

```
>>> box2d = Box2D(1, 2, 3, 4)
>>> box2d.ymax
4
```

property tl: *tensorbay.geometry.vector.Vector2D*

Return the top left point.

Returns The top left point.

Examples

```
>>> box2d = Box2D(1, 2, 3, 4)
>>> box2d.tl
Vector2D(1, 2)
```

property br: *tensorbay.geometry.vector.Vector2D*

Return the bottom right point.

Returns The bottom right point.

Examples

```
>>> box2d = Box2D(1, 2, 3, 4)
>>> box2d.br
Vector2D(3, 4)
```

property width: float

Return the width of the 2D box.

Returns The width of the 2D box.

Examples

```
>>> box2d = Box2D(1, 2, 3, 6)
>>> box2d.width
2
```

property height: float

Return the height of the 2D box.

Returns The height of the 2D box.

Examples

```
>>> box2d = Box2D(1, 2, 3, 6)
>>> box2d.height
4
```

dumps()
Dumps a 2D box into a dict.

Returns A dict containing vertex coordinates of the box.

Return type Dict[str, float]

Examples

```
>>> box2d = Box2D(1, 2, 3, 4)
>>> box2d.dumps()
{'xmin': 1, 'ymin': 2, 'xmax': 3, 'ymax': 4}
```

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

Returns The area of the 2D box.

Return type float

Examples

```
>>> box2d = Box2D(1, 2, 3, 4)
>>> box2d.area()
4
```

class tensorbay.geometry.box.**Box3D**(*size*, *translation=(0, 0, 0)*, *rotation=(1, 0, 0, 0)*, ***, *transform_matrix=None*)
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

- **translation** (*Iterable[float]*) – Translation in a sequence of [x, y, z].
- **rotation** (*Union[Iterable[float], quaternion.quaternion]*) – Rotation in a sequence of [w, x, y, z] or numpy quaternion.
- **size** (*Iterable[float]*) – Size in a sequence of [x, y, z].
- **transform_matrix** (*Optional[Union[Sequence[Sequence[float]], numpy.ndarray]]*) – A 4x4 or 3x4 transform matrix.

Return type None

Examples

Initialization Method 1: Init from size, translation and rotation.

```
>>> Box3D([1, 2, 3], [0, 1, 0, 0], [1, 2, 3])
Box3D(
  (size): Vector3D(1, 2, 3)
  (translation): Vector3D(1, 2, 3),
  (rotation): quaternion(0, 1, 0, 0),
)
```

Initialization Method 2: Init from size and transform matrix.

```
>>> from tensorbay.geometry import Transform3D
>>> matrix = [[1, 0, 0, 1], [0, 1, 0, 2], [0, 0, 1, 3]]
>>> Box3D(size=[1, 2, 3], transform_matrix=matrix)
Box3D(
  (size): Vector3D(1, 2, 3)
  (translation): Vector3D(1, 2, 3),
  (rotation): quaternion(1, -0, -0, -0),
)
```

classmethod loads(*contents*)

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

Parameters **contents** (*Mapping[str, Mapping[str, float]]*) – A dict containing the coordinates of a 3D box.

Returns The loaded *Box3D* object.

Return type tensorbay.geometry.box._B3

Examples

```
>>> contents = {
...     "size": {"x": 1.0, "y": 2.0, "z": 3.0},
...     "translation": {"x": 1.0, "y": 2.0, "z": 3.0},
...     "rotation": {"w": 0.0, "x": 1.0, "y": 0.0, "z": 0.0},
... }
>>> Box3D.loads(contents)
Box3D(
  (size): Vector3D(1.0, 2.0, 3.0)
  (translation): Vector3D(1.0, 2.0, 3.0),
  (rotation): quaternion(0, 1, 0, 0),
)
```

classmethod iou(*box1*, *box2*, *angle_threshold=5*)

Calculate the intersection over union between two 3D boxes.

Parameters

- **box1** (tensorbay.geometry.box.Box3D) – A 3D box.
- **box2** (tensorbay.geometry.box.Box3D) – A 3D box.
- **angle_threshold** (*float*) – The threshold of the relative angles between two input 3d boxes in degree.

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

Return type float

Examples

```
>>> box3d_1 = Box3D(size=[1, 1, 1])
>>> box3d_2 = Box3D(size=[2, 2, 2])
>>> Box3D.iou(box3d_1, box3d_2)
0.125
```

property translation: *tensorbay.geometry.vector.Vector3D*
Return the translation of the 3D box.

Returns The translation of the 3D box.

Examples

```
>>> box3d = Box3D(size=(1, 1, 1), translation=(1, 2, 3))
>>> box3d.translation
Vector3D(1, 2, 3)
```

property rotation: **quaternion.quaternion**
Return the rotation of the 3D box.

Returns The rotation of the 3D box.

Examples

```
>>> box3d = Box3D(size=(1, 1, 1), rotation=(0, 1, 0, 0))
>>> box3d.rotation
quaternion(0, 1, 0, 0)
```

property transform: *tensorbay.geometry.transform.Transform3D*
Return the transform of the 3D box.

Returns The transform of the 3D box.

Examples

```
>>> box3d = Box3D(size=(1, 1, 1), translation=(1, 2, 3), rotation=(1, 0, 0, 0))
>>> box3d.transform
Transform3D(
  (translation): Vector3D(1, 2, 3),
  (rotation): quaternion(1, 0, 0, 0)
)
```

property size: *tensorbay.geometry.vector.Vector3D*
Return the size of the 3D box.

Returns The size of the 3D box.

Examples

```
>>> box3d = Box3D(size=(1, 1, 1))
>>> box3d.size
Vector3D(1, 1, 1)
```

volume()
Return the volume of the 3D box.

Returns The volume of the 3D box.

Return type float

Examples

```
>>> box3d = Box3D(size=(1, 2, 3))
>>> box3d.volume()
6
```

dumps()
Dumps the 3D box into a dict.

Returns A dict containing translation, rotation and size information.

Return type Dict[str, Dict[str, float]]

Examples

```
>>> box3d = Box3D(size=(1, 2, 3), translation=(1, 2, 3), rotation=(0, 1, 0, 0))
>>> box3d.dumps()
{
    "translation": {"x": 1, "y": 2, "z": 3},
    "rotation": {"w": 0.0, "x": 1.0, "y": 0.0, "z": 0.0},
    "size": {"x": 1, "y": 2, "z": 3},
}
```

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*, ***kwargs*)
Bases: *tensorbay.geometry.vector.Vector2D*

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.

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

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

- **args** (*float*) –
- **kwargs** (*float*) –

Return type tensorbay.geometry.vector._V2

Examples

Initialization Method 1: Init from coordinates of x, y.

```
>>> Keypoint2D(1.0, 2.0)
Keypoint2D(1.0, 2.0)
```

Initialization Method 2: Init from coordinates and visible status.

```
>>> Keypoint2D(1.0, 2.0, 0)
Keypoint2D(1.0, 2.0, 0)
```

classmethod loads(*contents*)

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

Parameters **contents** (*Mapping[str, float]*) – A dict containing coordinates and visible status(optional) of a 2D keypoint.

Returns The loaded *Keypoint2D* object.

Return type tensorbay.geometry.keypoint._T

Examples

```
>>> contents = {"x":1.0,"y":2.0,"v":1}
>>> Keypoint2D.loads(contents)
Keypoint2D(1.0, 2.0, 1)
```

property v: Optional[int]

Return the visible status of the 2D keypoint.

Returns Visible status of the 2D keypoint.

Examples

```
>>> keypoint = Keypoint2D(3.0, 2.0, 1)
>>> keypoint.v
1
```

dumps()

Dumps the *Keypoint2D* into a dict.

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

Return type Dict[str, float]

Examples

```
>>> keypoint = Keypoint2D(1.0, 2.0, 1)
>>> keypoint.dumps()
{'x': 1.0, 'y': 2.0, 'v': 1}
```

class tensorbay.geometry.keypoint.**Keypoints2D**(*points=None*)

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

This class defines the concept of Keypoints2D.

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

Examples

```
>>> Keypoints2D([[1, 2], [2, 3]])
Keypoints2D [
  Keypoint2D(1, 2),
  Keypoint2D(2, 3)
]
```

classmethod loads(*contents*)

Load a *Keypoints2D* from a list of dict.

Parameters contents (*Sequence[Mapping[str, float]]*) – A list of dictionaries containing 2D keypoint.

Returns The loaded *Keypoints2D* object.

Return type tensorbay.geometry.keypoint._P

Examples

```
>>> contents = [{"x": 1.0, "y": 1.0, "v": 1}, {"x": 2.0, "y": 2.0, "v": 2}]
>>> Keypoints2D.loads(contents)
Keypoints2D [
  Keypoint2D(1.0, 1.0, 1),
  Keypoint2D(2.0, 2.0, 2)
]
```

tensorbay.geometry.point_list

PointList2D, MultiPointList2D.

PointList2D contains a list of 2D points.

MultiPointList2D contains multiple 2D point lists.

class tensorbay.geometry.point_list.**PointList2D**(*points=None*)

Bases: *tensorbay.utility.user.UserMutableSequence*[tensorbay.geometry.point_list._T]

This class defines the concept of PointList2D.

PointList2D contains a list of 2D points.

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

classmethod loads(*contents*)

Load a *PointList2D* from a list of dictionaries.

Parameters **contents** (*Sequence[Mapping[str, float]]*) – A list of dictionaries containing the coordinates of the vertexes of the point list:

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

Returns The loaded *PointList2D* object.

Return type tensorbay.geometry.point_list._P

dumps()

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.

Return type List[Dict[str, float]]

bounds()

Calculate the bounds of point list.

Returns The bounds of point list.

Return type *tensorbay.geometry.box.Box2D*

class tensorbay.geometry.point_list.**MultiPointList2D**(*point_lists=None*)

Bases: *tensorbay.utility.user.UserMutableSequence*[tensorbay.geometry.point_list._L]

This class defines the concept of MultiPointList2D.

MultiPointList2D contains multiple 2D point lists.

Parameters **point_lists** – A list of 2D point list.

classmethod loads(*contents*)

Loads a *MultiPointList2D* from the given contents.

Parameters **contents** (*Sequence[Sequence[Mapping[str, float]]]*) – A list of dictionary lists containing the coordinates of the vertexes of the multiple point lists:

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

Returns The loaded *MultiPointList2D* object.

Return type tensorbay.geometry.point_list._P

dumps()

Dumps all the information of the *MultiPointList2D*.

Returns All the information of the *MultiPointList2D*.

Return type List[List[Dict[str, float]]]

bounds()

Calculate the bounds of multiple point lists.

Returns The bounds of multiple point lists.

Return type *tensorbay.geometry.box.Box2D*

tensorbay.geometry.polygon

Polygon.

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

class tensorbay.geometry.polygon.**Polygon**(*points=None*)

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

This class defines the concept of Polygon.

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

Examples

```
>>> Polygon([[1, 2], [2, 3], [2, 2]])
Polygon [
  Vector2D(1, 2),
  Vector2D(2, 3),
  Vector2D(2, 2)
]
```

classmethod loads(*contents*)

Loads the information of *Polygon*.

Parameters contents (*Sequence[Mapping[str, float]]*) – A list of dictionary lists containing the coordinates of the vertexes of the polygon.

Returns The loaded *Polygon* object.

Return type tensorbay.geometry.polygon._P

Examples

```
>>> contents = [{"x": 1.0, "y": 1.0}, {"x": 2.0, "y": 2.0}, {"x": 2.0, "y": 3.0}
↪]
>>> Polygon.loads(contents)
Polygon [
  Vector2D(1.0, 1.0),
  Vector2D(2.0, 2.0),
  Vector2D(2.0, 3.0)
]
```

area()

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.

Return type float

Examples

```
>>> polygon = Polygon([[1, 2], [2, 2], [2, 3]])
>>> polygon.area()
0.5
```

class tensorbay.geometry.polygon.**MultiPolygon**(*polygons*)

Bases: *tensorbay.geometry.point_list.MultiPointList2D*[*tensorbay.geometry.polygon.Polygon*]

This class defines the concept of MultiPolygon.

MultiPolygon contains a list of polygons.

Parameters polygons – A list of polygons.

Examples

```
>>> MultiPolygon([[[1.0, 4.0], [2.0, 3.7], [7.0, 4.0]],
...               [[5.0, 7.0], [6.0, 7.0], [9.0, 8.0]]])
MultiPolygon [
    Polygon [...]
    Polygon [...]
    ...
]
```

classmethod loads(*contents*)

Loads a *MultiPolygon* from the given contents.

Parameters **contents** (*Sequence[Sequence[Mapping[str, float]]]*) – A list of dict lists containing the coordinates of the vertices of the polygon list.

Returns The loaded `MultiPolyline2D` object.

Return type tensorbay.geometry.polygon._P

Examples

```
>>> contents = [[{'x': 1.0, 'y': 4.0}, {'x': 2.0, 'y': 3.7}, {'x': 7.0, 'y': 4.
↪0}],
...             [{'x': 5.0, 'y': 7.0}, {'x': 6.0, 'y': 7.0}, {'x': 9.0, 'y': 8.
↪0}]]
>>> multipolygon = MultiPolygon.loads(contents)
>>> multipolygon
MultiPolygon [
    Polygon [...]
    Polygon [...]
    ...
]
```

dumps()

Dumps a *MultiPolygon* into a polygon list.

Returns All the information of the *MultiPolygon*.

Return type List[List[Dict[str, float]]]

Examples

```
>>> multipolygon = MultiPolygon([[[1.0, 4.0], [2.0, 3.7], [7.0, 4.0]],
...                              [[5.0, 7.0], [6.0, 7.0], [9.0, 8.0]]])
>>> multipolygon.dumps()
[
    [{'x': 1.0, 'y': 4.0}, {'x': 2.0, 'y': 3.7}, {'x': 7.0, 'y': 4.0}],
    [{'x': 5,0, 'y': 7.0}, {'x': 6.0, 'y': 7.0}, {'x': 9.0, 'y': 8.0}]
]
```

class tensorbay.geometry.polygon.**RLE**(*rle*)

Bases: *tensorbay.utility.user.UserMutableSequence*[int]

This class defines the concept of RLE.

RLE contains an rle format mask.

Parameters **rle** – A rle format mask.

Examples

```
>>> RLE([272, 2, 4, 4, 2, 9])
RLE [
  272,
  2,
  ...
]
```

classmethod loads(*contents*)

Loads a :class:RLE` from the given contents.

Parameters **contents** (*List[int]*) – One rle mask.

Returns The loaded *RLE* object.

Return type *tensorbay.geometry.polygon.RLE*

Examples

```
>>> contents = [272, 2, 4, 4, 2, 9]
>>> rle = RLE.loads(contents)
>>> rle
RLE [
  272,
  2,
  ...
]
```

dumps()

Dumps a *RLE* into one rle mask.

Returns All the information of the *RLE*.

Return type List[int]

Examples

```
>>> rle = RLE([272, 2, 4, 4, 2, 9])
>>> rle.dumps()
[272, 2, 4, 4, 2, 9]
```

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()*.

MultiPolyline2D contains a list of polyline.

class tensorbay.geometry.polyline.**Polyline2D**(*points=None*)

Bases: *tensorbay.geometry.point_list.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()*.

Examples

```
>>> Polyline2D([[1, 2], [2, 3]])
Polyline2D [
  Vector2D(1, 2),
  Vector2D(2, 3)
]
```

static uniform_frechet_distance(*polyline1*, *polyline2*)

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

Parameters

- **polyline1** (*Sequence[Sequence[float]]*) – The first polyline consists of multiple points.
- **polyline2** (*Sequence[Sequence[float]]*) – The second polyline consists of multiple points.

Returns The computed distance between the two polylines.

Return type float

Examples

```
>>> polyline_1 = [[1, 1], [1, 2], [2, 2]]
>>> polyline_2 = [[4, 5], [2, 1], [3, 3]]
>>> Polyline2D.uniform_frechet_distance(polyline_1, polyline_2)
3.605551275463989
```

static similarity(*polyline1*, *polyline2*)

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

Parameters

- **polyline1** (*Sequence[Sequence[float]]*) – The first polyline consists of multiple points.
- **polyline2** (*Sequence[Sequence[float]]*) – The second polyline consisting of multiple points.

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

Return type float

Examples

```
>>> polyline_1 = [[1, 1], [1, 2], [2, 2]]
>>> polyline_2 = [[4, 5], [2, 1], [3, 3]]
>>> Polyline2D.similarity(polyline_1, polyline_2)
0.2788897449072022
```

classmethod loads(*contents*)

Load a *Polyline2D* from a list of dict.

Parameters contents (*Sequence[Mapping[str, float]]*) – A list of dict containing the coordinates of the vertexes of the polyline.

Returns The loaded *Polyline2D* object.

Return type tensorbay.geometry.polyline._P

Examples

```
>>> polyline = Polyline2D([[1, 1], [1, 2], [2, 2]])
>>> polyline.dumps()
[{'x': 1, 'y': 1}, {'x': 1, 'y': 2}, {'x': 2, 'y': 2}]
```

class tensorbay.geometry.polyline.**MultiPolyline2D**(*polylines=None*)

Bases: *tensorbay.geometry.point_list.MultiPointList2D*[*tensorbay.geometry.polyline.Polyline2D*]

This class defines the concept of MultiPolyline2D.

MultiPolyline2D contains a list of polylines.

Parameters polylines – A list of polylines.

Examples

```
>>> MultiPolyline2D([[[1, 2], [2, 3]], [[3, 4], [6, 8]]])
MultiPolyline2D [
    Polyline2D [...]
    Polyline2D [...]
    ...
]
```

classmethod loads(*contents*)

Loads a *MultiPolyline2D* from the given contents.

Parameters contents (*Sequence[Sequence[Mapping[str, float]]]*) – A list of dict lists containing the coordinates of the vertexes of the polyline list.

Returns The loaded *MultiPolyline2D* object.

Return type tensorbay.geometry.polyline._P

Examples

```
>>> contents = [[{'x': 1, 'y': 1}, {'x': 1, 'y': 2}, {'x': 2, 'y': 2}],
                [{'x': 2, 'y': 3}, {'x': 3, 'y': 5}]]
>>> multipolyline = MultiPolyline2D.loads(contents)
>>> multipolyline
MultiPolyline2D [
    Polyline2D [...]
    Polyline2D [...]
    ...
]
```

dumps()

Dumps a *MultiPolyline2D* into a polyline list.

Returns All the information of the *MultiPolyline2D*.

Return type List[List[Dict[str, float]]]

Examples

```
>>> multipolyline = MultiPolyline2D([[[1, 1], [1, 2], [2, 2]], [[2, 3], [3,␣
↪5]]])
>>> multipolyline.dumps()
[
    [{'x': 1, 'y': 1}, {'x': 1, 'y': 2}, {'x': 2, 'y': 2}],
    [{'x': 2, 'y': 3}, {'x': 3, 'y': 5}]
]
```

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**(*translation=(0, 0, 0)*, *rotation=(1, 0, 0, 0)*, ***, *matrix=None*)

Bases: *tensorbay.utility.repr.ReprMixin*

This class defines the concept of Transform3D.

Transform3D contains rotation and translation of the 3D transform.

Parameters

- **translation** (*Iterable[float]*) – Translation in a sequence of [x, y, z].
- **rotation** (*Union[Iterable[float], quaternion.quaternion]*) – Rotation in a sequence of [w, x, y, z] or numpy quaternion.
- **matrix** (*Optional[Union[Sequence[Sequence[float]], numpy.ndarray]]*) – A 4x4 or 3x4 transform matrix.

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

Return type None

Examples

Initialization Method 1: Init from translation and rotation.

```
>>> Transform3D([1, 1, 1], [1, 0, 0, 0])
Transform3D(
  (translation): Vector3D(1, 1, 1),
  (rotation): quaternion(1, 0, 0, 0)
)
```

Initialization Method 2: Init from transform matrix in sequence.

```
>>> Transform3D(matrix=[[1, 0, 0, 1], [0, 1, 0, 1], [0, 0, 1, 1]])
Transform3D(
  (translation): Vector3D(1, 1, 1),
  (rotation): quaternion(1, -0, -0, -0)
)
```

Initialization Method 3: Init from transform matrix in numpy array.

```
>>> import numpy as np
>>> Transform3D(matrix=np.array([[1, 0, 0, 1], [0, 1, 0, 1], [0, 0, 1, 1]]))
Transform3D(
  (translation): Vector3D(1, 1, 1),
  (rotation): quaternion(1, -0, -0, -0)
)
```

classmethod loads(*contents*)

Load a *Transform3D* from a dict containing rotation and translation.

Parameters **contents** (*Mapping[str, Mapping[str, float]]*) – A dict containing rotation and translation of a 3D transform.

Returns The loaded *Transform3D* object.

Return type tensorbay.geometry.transform._T

Example

```
>>> contents = {
...     "translation": {"x": 1.0, "y": 2.0, "z": 3.0},
...     "rotation": {"w": 1.0, "x": 0.0, "y": 0.0, "z": 0.0},
... }
>>> Transform3D.loads(contents)
Transform3D(
  (translation): Vector3D(1.0, 2.0, 3.0),
  (rotation): quaternion(1, 0, 0, 0)
)
```

property translation: *tensorbay.geometry.vector.Vector3D*

Return the translation of the 3D transform.

Returns Translation in *Vector3D*.

Examples

```
>>> transform = Transform3D(matrix=[[1, 0, 0, 1], [0, 1, 0, 1], [0, 0, 1, 1]])
>>> transform.translation
Vector3D(1, 1, 1)
```

property rotation: quaternion.quaternion
Return the rotation of the 3D transform.

Returns Rotation in numpy quaternion.

Examples

```
>>> transform = Transform3D(matrix=[[1, 0, 0, 1], [0, 1, 0, 1], [0, 0, 1, 1]])
>>> transform.rotation
quaternion(1, -0, -0, -0)
```

dumps()
Dumps the *Transform3D* into a dict.

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

Return type Dict[str, Dict[str, float]]

Examples

```
>>> transform = Transform3D(matrix=[[1, 0, 0, 1], [0, 1, 0, 1], [0, 0, 1, 1]])
>>> transform.dumps()
{
    'translation': {'x': 1, 'y': 1, 'z': 1},
    'rotation': {'w': 1.0, 'x': -0.0, 'y': -0.0, 'z': -0.0},
}
```

set_translation(*x*, *y*, *z*)
Set the translation of the transform.

Parameters

- **x** (*float*) – The x coordinate of the translation.
- **y** (*float*) – The y coordinate of the translation.
- **z** (*float*) – The z coordinate of the translation.

Return type None

Examples

```
>>> transform = Transform3D([1, 1, 1], [1, 0, 0, 0])
>>> transform.set_translation(3, 4, 5)
>>> transform
Transform3D(
  (translation): Vector3D(3, 4, 5),
  (rotation): quaternion(1, 0, 0, 0)
)
```

set_rotation(*w=None*, *x=None*, *y=None*, *z=None*, *, *quaternion=None*)
Set the rotation of the transform.

Parameters

- **w** (*Optional[float]*) – The w componet of the roation quaternion.
- **x** (*Optional[float]*) – The x componet of the roation quaternion.
- **y** (*Optional[float]*) – The y componet of the roation quaternion.
- **z** (*Optional[float]*) – The z componet of the roation quaternion.
- **quaternion** (*Optional[quaternion.quaternion]*) – Numpy quaternion representing the rotation.

Return type None

Examples

```
>>> transform = Transform3D([1, 1, 1], [1, 0, 0, 0])
>>> transform.set_rotation(0, 1, 0, 0)
>>> transform
Transform3D(
  (translation): Vector3D(1, 1, 1),
  (rotation): quaternion(0, 1, 0, 0)
)
```

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

Returns A 4x4 numpy array represents the transform matrix.

Return type numpy.ndarray

Examples

```
>>> transform = Transform3D([1, 2, 3], [0, 1, 0, 0])
>>> transform.as_matrix()
array([[ 1.,  0.,  0.,  1.],
       [ 0., -1.,  0.,  2.],
       [ 0.,  0., -1.,  3.],
       [ 0.,  0.,  0.,  1.]])
```

inverse()
Return the inverse of the transform.

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

Parameters **self** (*tensorbay.geometry.transform._T*) –

Return type tensorbay.geometry.transform._T

Examples

```
>>> transform = Transform3D([1, 2, 3], [0, 1, 0, 0])
>>> transform.inverse()
Transform3D(
  (translation): Vector3D(-1.0, 2.0, 3.0),
  (rotation): quaternion(0, -1, -0, -0)
)
```

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*, *y*, *z=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** (*float*) – The x coordinate of the vector.
- **y** (*float*) – The y coordinate of the vector.
- **z** (*Optional[float]*) – The z coordinate of the vector.

Return type Union[*Vector2D*, *Vector3D*]

Examples

```
>>> Vector(1, 2)
Vector2D(1, 2)
```

```
>>> Vector(1, 2, 3)
Vector3D(1, 2, 3)
```

static loads(*contents*)

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

Parameters **contents** (*Mapping[str, float]*) – A dict containing coordinates of the vector.

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

Return type Union[*tensorbay.geometry.vector.Vector2D*, *tensorbay.geometry.vector.Vector3D*]

Examples

```
>>> contents = {"x": 1.0, "y": 2.0}
>>> Vector.loads(contents)
Vector2D(1.0, 2.0)
```

```
>>> contents = {"x": 1.0, "y": 2.0, "z": 3.0}
>>> Vector.loads(contents)
Vector3D(1.0, 2.0, 3.0)
```

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

Bases: `tensorbay.geometry.vector.Vector`

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.
- **args** (*float*) –
- **kwargs** (*float*) –

Return type tensorbay.geometry.vector._V2

Examples

```
>>> Vector2D(1, 2)
Vector2D(1, 2)
```

classmethod loads(*contents*)

Load a `Vector2D` object from a dict containing coordinates of a 2D vector.

Parameters **contents** (*Mapping[str, float]*) – A dict containing coordinates of a 2D vector.

Returns The loaded `Vector2D` object.

Return type tensorbay.geometry.vector._V2

Examples

```
>>> contents = {"x": 1.0, "y": 2.0}
>>> Vector2D.loads(contents)
Vector2D(1.0, 2.0)
```

property x: float

Return the x coordinate of the vector.

Returns X coordinate in float type.

Examples

```
>>> vector_2d = Vector2D(1, 2)
>>> vector_2d.x
1
```

property y: float

Return the y coordinate of the vector.

Returns Y coordinate in float type.

Examples

```
>>> vector_2d = Vector2D(1, 2)
>>> vector_2d.y
2
```

dumps()

Dumps the vector into a dict.

Returns A dict containing the vector coordinate.

Return type Dict[str, float]

Examples

```
>>> vector_2d = Vector2D(1, 2)
>>> vector_2d.dumps()
{'x': 1, 'y': 2}
```

class tensorbay.geometry.vector.**Vector3D**(**args*, ***kwargs*)

Bases: *tensorbay.geometry.vector.Vector*

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.
- **args** (*float*) –
- **kwargs** (*float*) –

Return type tensorbay.geometry.vector._V3

Examples

```
>>> Vector3D(1, 2, 3)
Vector3D(1, 2, 3)
```

classmethod loads(*contents*)

Load a *Vector3D* object from a dict containing coordinates of a 3D vector.

Parameters **contents** (*Mapping[str, float]*) – A dict contains coordinates of a 3D vector.

Returns The loaded *Vector3D* object.

Return type tensorbay.geometry.vector._V3

Examples

```
>>> contents = {"x": 1.0, "y": 2.0, "z": 3.0}
>>> Vector3D.loads(contents)
Vector3D(1.0, 2.0, 3.0)
```

property x: float

Return the x coordinate of the vector.

Returns X coordinate in float type.

Examples

```
>>> vector_3d = Vector3D(1, 2, 3)
>>> vector_3d.x
1
```

property y: float

Return the y coordinate of the vector.

Returns Y coordinate in float type.

Examples

```
>>> vector_3d = Vector3D(1, 2, 3)
>>> vector_3d.y
2
```

property z: float

Return the z coordinate of the vector.

Returns Z coordinate in float type.

Examples

```
>>> vector_3d = Vector3D(1, 2, 3)
>>> vector_3d.z
3
```

dumps()

Dumps the vector into a dict.

Returns A dict containing the vector coordinates.

Return type Dict[str, float]

Examples

```
>>> vector_3d = Vector3D(1, 2, 3)
>>> vector_3d.dumps()
{'x': 1, 'y': 2, 'z': 3}
```

1.24.4 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.**Items**(*, *type_=''*, *enum=None*, *minimum=None*, *maximum=None*, *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.

Todo: The format of argument *type_* on the generated web page is incorrect.

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** (*Optional[Iterable[Optional[Union[str, float, bool]]]]*) – All the possible values of an enumeration attribute.
- **minimum** (*Optional[float]*) – The minimum value of number type attribute.
- **maximum** (*Optional[float]*) – The maximum value of number type attribute.
- **items** (*Optional[Items]*) – The items inside array type attributes.
- **type_** (*Union[str, None, Type[Optional[Union[list, bool, int, float, str]]], Iterable[Union[str, None, Type[Optional[Union[list, bool, int, float, str]]]]]]*) –

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.

Parameters

- **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]]]]*) –
- **minimum** (*Optional[float]*) –
- **maximum** (*Optional[float]*) –
- **items** (*Optional[Items]*) –

Examples

```
>>> Items(type_="integer", enum=[1, 2, 3, 4, 5], minimum=1, maximum=5)
Items(
  (type): 'integer',
  (enum): [...],
  (minimum): 1,
  (maximum): 5
)
```

classmethod loads(*contents*)
Load an Items from a dict containing the items information.

Parameters contents (*Dict[str, Any]*) – A dict containing the information of the items.

Returns The loaded *Items* object.

Return type tensorbay.label.attributes._T

Examples

```
>>> contents = {
...     "type": "array",
...     "enum": [1, 2, 3, 4, 5],
...     "minimum": 1,
...     "maximum": 5,
...     "items": {
...         "enum": [None],
...         "type": "null",
...     },
... }
>>> Items.loads(contents)
Items(
  (type): 'array',
  (enum): [...],
  (minimum): 1,
  (maximum): 5,
  (items): Items(...)
)
```

dumps()

Dumps the information of the items into a dict.

Returns A dict containing all the information of the items.

Return type Dict[str, Any]

Examples

```
>>> items = Items(type_="integer", enum=[1, 2, 3, 4, 5], minimum=1, maximum=5)
>>> items.dumps()
{'type': 'integer', 'enum': [1, 2, 3, 4, 5], 'minimum': 1, 'maximum': 5}
```

class tensorbay.label.attributes.**AttributeInfo**(*name*, ***, *type_=''*, *enum=None*, *minimum=None*, *maximum=None*, *items=None*, *parent_categories=None*, *description=''*)

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.

Todo: The format of argument *type_* on the generated web page is incorrect.

Parameters

- **name** (*str*) – 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** (*Optional[Iterable[Optional[Union[str, float, bool]]]]*) – All the possible values of an enumeration attribute.
- **minimum** (*Optional[float]*) – The minimum value of number type attribute.
- **maximum** (*Optional[float]*) – The maximum value of number type attribute.
- **items** (*Optional[*tensorbay.label.attributes.Items*]*) – The items inside array type attributes.
- **parent_categories** (*List[str]*) – The parent categories of the attribute.
- **description** (*str*) – The description of the attribute.
- **type_** (*Union[str, None, Type[Optional[Union[list, bool, int, float, str]]], Iterable[Union[str, None, Type[Optional[Union[list, bool, int, float, str]]]]]]*) –

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.

Type List[str]

description
The description of the attribute.

Type str

Examples

```
>>> from tensorbay.label import Items
>>> items = Items(type_="integer", enum=[1, 2, 3, 4, 5], minimum=1, maximum=5)
>>> AttributeInfo(
...     name="example",
...     type_="array",
...     enum=[1, 2, 3, 4, 5],
...     items=items,
...     minimum=1,
...     maximum=5,
...     parent_categories=["parent_category_of_example"],
...     description="This is an example",
... )
AttributeInfo("example")(
  (type): 'array',
  (enum): [
    1,
    2,
    3,
    4,
    5
  ],
  (minimum): 1,
  (maximum): 5,
  (items): Items(
    (type): 'integer',
    (enum): [...],
    (minimum): 1,
    (maximum): 5
  ),
  (parent_categories): [
    'parent_category_of_example'
  ]
)
```

classmethod loads(*contents*)

Load an AttributeInfo from a dict containing the attribute information.

Parameters **contents** (*Dict[str, Any]*) – A dict containing the information of the attribute.

Returns The loaded *AttributeInfo* object.

Return type tensorbay.label.attributes._T

Examples

```
>>> contents = {
...     "name": "example",
...     "type": "array",
...     "items": {"type": "boolean"},
...     "description": "This is an example",
...     "parentCategories": ["parent_category_of_example"],
... }
>>> AttributeInfo.loads(contents)
AttributeInfo("example")(
  (type): 'array',
  (items): Items(
    (type): 'boolean',
  ),
  (parent_categories): [
    'parent_category_of_example'
  ]
)
```

dumps()

Dumps the information of this attribute into a dict.

Returns A dict containing all the information of this attribute.

Return type Dict[str, Any]

Examples

```
>>> from tensorbay.label import Items
>>> items = Items(type_="integer", minimum=1, maximum=5)
>>> attributeinfo = AttributeInfo(
...     name="example",
...     type_="array",
...     items=items,
...     parent_categories=["parent_category_of_example"],
...     description="This is an example",
... )
>>> attributeinfo.dumps()
{
    'name': 'example',
    'description': 'This is an example',
    'type': 'array',
    'items': {'type': 'integer', 'minimum': 1, 'maximum': 5},
    'parentCategories': ['parent_category_of_example'],
}
```

tensorbay.label.basic

SubcatalogBase.

`Subcatalogbase` is the base class for different types of subcatalogs, which defines the basic concept of Subcatalog.

A subcatalog class extends *SubcatalogBase* and needed `SubcatalogMixin` classes.

class tensorbay.label.basic.**SubcatalogBase**(*description=''*)

Bases: *tensorbay.utility.repr.ReprMixin*, *tensorbay.utility.attr.AttrsMixin*

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.

Parameters **description** (*str*) – The description of the entire subcatalog.

Return type None

description

The description of the entire subcatalog.

Type str

classmethod loads(*contents*)

Loads a subcatalog from a dict containing the information of the subcatalog.

Parameters **contents** (*Dict[str, Any]*) – A dict containing the information of the subcatalog.

Returns The loaded *SubcatalogBase* object.

Return type tensorbay.label.basic._T

dumps()

Dumps all the information of the subcatalog into a dict.

Returns A dict containing all the information of the subcatalog.

Return type Dict[str, Any]

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 1.8: 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 keypoints type of label
PolygonSubcatalog	subcatalog for polygon type of label
Polyline2DSubcatalog	subcatalog for 2D polyline type of label
MultiPolygonSubcatalog	subcatalog for multiple polygon type of label
RLESubcatalog	subcatalog for rle mask type of label
MultiPolyline2DSubcatalog	subcatalog for 2D multiple polyline type of label
SentenceSubcatalog	subcatalog for transcripted sentence type of label

class `tensorbay.label.catalog.`**`Catalog`**

Bases: *tensorbay.utility.repr.ReprMixin*, *tensorbay.utility.attr.AttrsMixin*

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.

Examples

```
>>> from tensorbay.utility import NameList
>>> from tensorbay.label import ClassificationSubcatalog, CategoryInfo
>>> classification_subcatalog = ClassificationSubcatalog()
>>> categories = NameList()
>>> categories.append(CategoryInfo("example"))
>>> classification_subcatalog.categories = categories
>>> catalog = Catalog()
>>> catalog.classification = classification_subcatalog
>>> catalog
Catalog(
  (classification): ClassificationSubcatalog(
    (categories): NameList [...]
  )
)
```

classmethod loads(*contents*)

Load a Catalog from a dict containing the catalog information.

Parameters **contents** (*Dict[str, Any]*) – A dict containing all the information of the catalog.

Returns The loaded *Catalog* object.

Return type tensorbay.label.catalog._T

Examples

```
>>> contents = {
...     "CLASSIFICATION": {
...         "categories": [
...             {
...                 "name": "example",
...             }
...         ]
...     },
...     "KEYPOINTS2D": {
...         "keypoints": [
...             {
...                 "number": 5,
...             }
...         ]
...     },
... }
>>> Catalog.loads(contents)
Catalog(
  (classification): ClassificationSubcatalog(
    (categories): NameList [...]
  ),
  (keypoints2d): Keypoints2DSubcatalog(
    (is_tracking): False,
    (keypoints): [...]
  )
)
```

dumps()

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.

Return type Dict[str, Any]

Examples

```
>>> # catalog is the instance initialized above.
>>> catalog.dumps()
{'CLASSIFICATION': {'categories': [{'name': 'example'}]}}
```

tensorbay.label.label

Label.

A `Data` instance contains one or several types of labels, all of which are stored in `label`.

Different label types correspond to different label classes classes.

Table 1.9: label classes

label classes	explaination
Classification	classification type of label
LabeledBox2D	2D bounding box type of label
LabeledBox3D	3D bounding box type of label
LabeledPolygon	polygon type of label
LabeledMultiPolygon	polygon lists type of label
LabeledRLE	rle mask type of label
LabeledPolyline2D	2D polyline type of label
LabeledMultiPolyline2D	2D polyline lists type of label
LabeledKeypoints2D	2D keypoints type of label
LabeledSentence	transcripted sentence type of label

class tensorbay.label.label.**Label**

Bases: *tensorbay.utility.repr.ReprMixin*, *tensorbay.utility.attr.AttrsMixin*

This class defines *label*.

It contains growing types of labels referring to different tasks.

Examples

```
>>> from tensorbay.label import Classification
>>> label = Label()
>>> label.classification = Classification("example_category", {"example_attribute1
↪": "a"})
>>> label
Label(
  (classification): Classification(
    (category): 'example_category',
    (attributes): {...}
  )
)
```

classmethod loads(*contents*)

Loads data from a dict containing the labels information.

Parameters contents (*Dict[str, Any]*) – A dict containing the labels information.

Returns A *Label* instance containing labels information from the given dict.

Return type tensorbay.label.label._T

Examples

```
>>> contents = {
...     "CLASSIFICATION": {
...         "category": "example_category",
...         "attributes": {"example_attribute1": "a"}
...     }
... }
>>> Label.loads(contents)
```

(continues on next page)

(continued from previous page)

```
Label(
  (classification): Classification(
    (category): 'example_category',
    (attributes): {...}
  )
)
```

dumps()

Dumps all labels into a dict.

Returns Dumped labels dict.

Return type Dict[str, Any]

Examples

```
>>> from tensorbay.label import Classification
>>> label = Label()
>>> label.classification = Classification("category1", {"attribute1": "a"})
>>> label.dumps()
{'CLASSIFICATION': {'category': 'category1', 'attributes': {'attribute1': 'a'}}}
```

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=False*)

Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.IsTrackingMixin*, *tensorbay.label.supports.CategoriesMixin*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for 2D box type of labels.

Parameters **is_tracking** (*bool*) – A boolean value indicates whether the corresponding subcatalog contains tracking information.

Return type None

description

The description of the entire 2D box subcatalog.

Type str

categories

All the possible categories in the corresponding dataset stored in a *NameList* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameList*[*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 *NameList* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking
Whether the Subcatalog contains tracking information.

Type bool

Examples

Initialization Method 1: Init from Box2DSubcatalog.loads() method.

```
>>> catalog = {
...     "BOX2D": {
...         "isTracking": True,
...         "categoryDelimiter": ".",
...         "categories": [{"name": "0"}, {"name": "1"}],
...         "attributes": [{"name": "gender", "enum": ["male", "female"]}],
...     }
... }
>>> Box2DSubcatalog.loads(catalog["BOX2D"])
Box2DSubcatalog(
  (is_tracking): True,
  (category_delimiter): '.',
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty Box2DSubcatalog and then add the attributes.

```
>>> from tensorbay.utility import NameList
>>> from tensorbay.label import CategoryInfo, AttributeInfo
>>> categories = NameList()
>>> categories.append(CategoryInfo("a"))
>>> attributes = NameList()
>>> attributes.append(AttributeInfo("gender", enum=["female", "male"]))
>>> box2d_subcatalog = Box2DSubcatalog()
>>> box2d_subcatalog.is_tracking = True
>>> box2d_subcatalog.category_delimiter = "."
>>> box2d_subcatalog.categories = categories
>>> box2d_subcatalog.attributes = attributes
>>> box2d_subcatalog
Box2DSubcatalog(
  (is_tracking): True,
  (category_delimiter): '.',
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class tensorbay.label.label_box.**LabeledBox2D**(*xmin*, *ymin*, *xmax*, *ymax*, ***, *category=None*, *attributes=None*, *instance=None*)

Bases: `tensorbay.label.basic._LabelBase`, `tensorbay.geometry.box.Box2D`

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, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

instance

The instance id of the label.

Type str

Examples

```
>>> xmin, ymin, xmax, ymax = 1, 2, 4, 4
>>> LabeledBox2D(
...     xmin,
...     ymin,
...     xmax,
...     ymax,
...     category="example",
...     attributes={"attr": "a"},
...     instance="12345",
... )
LabeledBox2D(1, 2, 4, 4)(
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

classmethod from_xywh(*x*, *y*, *width*, *height*, ***, *category=None*, *attributes=None*, *instance=None*)

Create a `LabeledBox2D` instance from the top-left vertex, the width and height.

Parameters

- **x** (*float*) – X coordinate of the top left vertex of the box.
- **y** (*float*) – Y coordinate of the top left vertex of the box.
- **width** (*float*) – Length of the box along the x axis.
- **height** (*float*) – Length of the box along the y axis.
- **category** (*Optional[str]*) – The category of the label.
- **attributes** (*Optional[Dict[str, Any]]*) – The attributs of the label.
- **instance** (*Optional[str]*) – The instance id of the label.

Returns The created *LabeledBox2D* instance.

Return type tensorbay.label.label_box._T

Examples

```
>>> x, y, width, height = 1, 2, 3, 4
>>> LabeledBox2D.from_xywh(
...     x,
...     y,
...     width,
...     height,
...     category="example",
...     attributes={"key": "value"},
...     instance="12345",
... )
LabeledBox2D(1, 2, 4, 6)(
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

classmethod loads(*contents*)

Loads a LabeledBox2D from a dict containing the information of the label.

Parameters contents (*Mapping[str, Any]*) – A dict containing the information of the 2D bounding box label.

Returns The loaded *LabeledBox2D* object.

Return type tensorbay.label.label_box._T

Examples

```
>>> contents = {
...     "box2d": {"xmin": 1, "ymin": 2, "xmax": 5, "ymax": 8},
...     "category": "example",
...     "attributes": {"key": "value"},
...     "instance": "12345",
... }
>>> LabeledBox2D.loads(contents)
LabeledBox2D(1, 2, 5, 8)(
  (category): 'example',
```

(continues on next page)

(continued from previous page)

```
  (attributes): {...},
  (instance): '12345'
)
```

dumps()

Dumps the current 2D bounding box label into a dict.

Returns A dict containing all the information of the 2D box label.

Return type Dict[str, Any]

Examples

```
>>> xmin, ymin, xmax, ymax = 1, 2, 4, 4
>>> labelbox2d = LabeledBox2D(
...     xmin,
...     ymin,
...     xmax,
...     ymax,
...     category="example",
...     attributes={"attr": "a"},
...     instance="12345",
... )
>>> labelbox2d.dumps()
{
    'category': 'example',
    'attributes': {'attr': 'a'},
    'instance': '12345',
    'box2d': {'xmin': 1, 'ymin': 2, 'xmax': 4, 'ymax': 4},
}
```

class tensorbay.label.label_box.**Box3DSubcatalog**(*is_tracking=False*)

Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.IsTrackingMixin*, *tensorbay.label.supports.CategoriesMixin*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for 3D box type of labels.

Parameters **is_tracking** (*bool*) – A boolean value indicates whether the corresponding subcatalog contains tracking information.

Return type None

description

The description of the entire 3D box subcatalog.

Type str

categories

All the possible categories in the corresponding dataset stored in a `NameList` with the category names as keys and the `CategoryInfo` as values.

Type *tensorbay.utility.name.NameList*[*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 *NameList* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking
Whether the Subcatalog contains tracking information.

Type bool

Examples

Initialization Method 1: Init from Box3DSubcatalog.loads() method.

```
>>> catalog = {
...     "BOX3D": {
...         "isTracking": True,
...         "categoryDelimiter": ".",
...         "categories": [{"name": "0"}, {"name": "1"}],
...         "attributes": [{"name": "gender", "enum": ["male", "female"]}],
...     }
... }
>>> Box3DSubcatalog.loads(catalog["BOX3D"])
Box3DSubcatalog(
  (is_tracking): True,
  (category_delimiter): '.',
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty Box3DSubcatalog and then add the attributes.

```
>>> from tensorbay.utility import NameList
>>> from tensorbay.label import CategoryInfo, AttributeInfo
>>> categories = NameList()
>>> categories.append(CategoryInfo("a"))
>>> attributes = NameList()
>>> attributes.append(AttributeInfo("gender", enum=["female", "male"]))
>>> box3d_subcatalog = Box3DSubcatalog()
>>> box3d_subcatalog.is_tracking = True
>>> box3d_subcatalog.category_delimiter = "."
>>> box3d_subcatalog.categories = categories
>>> box3d_subcatalog.attributes = attributes
>>> box3d_subcatalog
Box3DSubcatalog(
  (is_tracking): True,
  (category_delimiter): '.',
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class tensorbay.label.label_box.**LabeledBox3D**(*size*, *translation=(0, 0, 0)*, *rotation=(1, 0, 0, 0)*, ***, *transform_matrix=None*, *category=None*, *attributes=None*, *instance=None*)

Bases: tensorbay.label.basic._LabelBase, *tensorbay.geometry.box.Box3D*

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

- **size** (*Iterable[float]*) – Size of the 3D bounding box label in a sequence of [x, y, z].
- **translation** (*Iterable[float]*) – Translation of the 3D bounding box label in a sequence of [x, y, z].
- **rotation** (*Union[Iterable[float], quaternion.quaternion]*) – Rotation of the 3D bounding box label in a sequence of [w, x, y, z] or a numpy quaternion object.
- **transform_matrix** (*Optional[Union[Sequence[Sequence[float]], numpy.ndarray]]*) – A 4x4 or 3x4 transformation matrix.
- **category** (*str*) – Category of the 3D bounding box label.
- **attributes** (*Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]*) – Attributs of the 3D bounding box label.
- **instance** (*str*) – 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, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

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.

Examples

```
>>> LabeledBox3D(
...     size=[1, 2, 3],
...     translation=(1, 2, 3),
...     rotation=(0, 1, 0, 0),
...     category="example",
...     attributes={"key": "value"},
...     instance="12345",
... )
LabeledBox3D(
```

(continues on next page)

(continued from previous page)

```
  (size): Vector3D(1, 2, 3),
  (translation): Vector3D(1, 2, 3),
  (rotation): quaternion(0, 1, 0, 0),
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

classmethod loads(*contents*)

Loads a LabeledBox3D from a dict containing the information of the label.

Parameters contents (*Mapping[str, Any]*) – A dict containing the information of the 3D bounding box label.

Returns The loaded *LabeledBox3D* object.

Return type tensorbay.label.label_box._T

Examples

```
>>> contents = {
...     "box3d": {
...         "size": {"x": 1, "y": 2, "z": 3},
...         "translation": {"x": 1, "y": 2, "z": 3},
...         "rotation": {"w": 1, "x": 0, "y": 0, "z": 0},
...     },
...     "category": "test",
...     "attributes": {"key": "value"},
...     "instance": "12345",
... }
>>> LabeledBox3D.loads(contents)
LabeledBox3D(
  (size): Vector3D(1, 2, 3),
  (translation): Vector3D(1, 2, 3),
  (rotation): quaternion(1, 0, 0, 0),
  (category): 'test',
  (attributes): {...},
  (instance): '12345'
)
```

dumps()

Dumps the current 3D bounding box label into a dict.

Returns A dict containing all the information of the 3D bounding box label.

Return type Dict[str, Any]

Examples

```
>>> labeledbox3d = LabeledBox3D(
...     size=[1, 2, 3],
...     translation=(1, 2, 3),
...     rotation=(0, 1, 0, 0),
...     category="example",
...     attributes={"key": "value"},
...     instance="12345",
... )
>>> labeledbox3d.dumps()
{
    'category': 'example',
    'attributes': {'key': 'value'},
    'instance': '12345',
    'box3d': {
        'translation': {'x': 1, 'y': 2, 'z': 3},
        'rotation': {'w': 0.0, 'x': 1.0, 'y': 0.0, 'z': 0.0},
        'size': {'x': 1, 'y': 2, 'z': 3},
    },
}
```

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.`**`ClassificationSubcatalog`**(*description=''*)

Bases: `tensorbay.label.basic.SubcatalogBase`, `tensorbay.label.supports.CategoriesMixin`, `tensorbay.label.supports.AttributesMixin`

This class defines the subcatalog for classification type of labels.

Parameters **description** (*str*) –

Return type None

description

The description of the entire classification subcatalog.

Type str

categories

All the possible categories in the corresponding dataset stored in a `NameList` with the category names as keys and the `CategoryInfo` as values.

Type *tensorbay.utility.name.NameList*[*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 *NameList* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

Examples

Initialization Method 1: Init from ClassificationSubcatalog.loads() method.

```
>>> catalog = {
...     "CLASSIFICATION": {
...         "categoryDelimiter": ".",
...         "categories": [
...             {"name": "a"},
...             {"name": "b"},
...         ],
...          "attributes": [{"name": "gender", "enum": ["male", "female"]}],
...     }
... }
>>> ClassificationSubcatalog.loads(catalog["CLASSIFICATION"])
ClassificationSubcatalog(
  (category_delimiter): '.',
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty ClassificationSubcatalog and then add the attributes.

```
>>> from tensorbay.utility import NameList
>>> from tensorbay.label import CategoryInfo, AttributeInfo, KeypointsInfo
>>> categories = NameList()
>>> categories.append(CategoryInfo("a"))
>>> attributes = NameList()
>>> attributes.append(AttributeInfo("gender", enum=["female", "male"]))
>>> classification_subcatalog = ClassificationSubcatalog()
>>> classification_subcatalog.category_delimiter = "."
>>> classification_subcatalog.categories = categories
>>> classification_subcatalog.attributes = attributes
>>> classification_subcatalog
ClassificationSubcatalog(
  (category_delimiter): '.',
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class tensorbay.label.label_classification.**Classification**(*category=None*, *attributes=None*)
Bases: tensorbay.label.basic._LabelBase

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** (*str*) – The category of the label.
- **attributes** (*Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]*) – The attributes of the label.

category
The category of the label.

Type str

attributes
The attributes of the label.

Type Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

Examples

```
>>> Classification(category="example", attributes={"attr": "a"})
Classification(
  (category): 'example',
  (attributes): {...}
)
```

classmethod loads(*contents*)
Loads a Classification label from a dict containing the label information.

Parameters **contents** (*Dict[str, Any]*) – A dict containing the information of the classification label.

Returns The loaded *Classification* object.

Return type tensorbay.label.label_classification._T

Examples

```
>>> contents = {"category": "example", "attributes": {"key": "value"}}
>>> Classification.loads(contents)
Classification(
  (category): 'example',
  (attributes): {...}
)
```

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=False*)
Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.IsTrackingMixin*, *tensorbay.label.supports.CategoriesMixin*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for 2D keypoints type of labels.

Parameters **is_tracking** (*bool*) – A boolean value indicates whether the corresponding subcatalog contains tracking information.

Return type None

description
The description of the entire 2D keypoints subcatalog.

Type str

categories
All the possible categories in the corresponding dataset stored in a *NameList* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameList*[*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 *NameList* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking
Whether the Subcatalog contains tracking information.

Type bool

Examples

Initialization Method 1: Init from Keypoints2DSubcatalog.loads() method.

```
>>> catalog = {
...     "KEYPOINTS2D": {
...         "isTracking": True,
...         "categories": [{"name": "0"}, {"name": "1"}],
...         "attributes": [{"name": "gender", "enum": ["male", "female"]}],
...         "keypoints": [
...             {
...                 "number": 2,
...                  "names": ["L_shoulder", "R_Shoulder"],
...                  "skeleton": [(0, 1)],
...             }
...         ],
...     }
... }
>>> Keypoints2DSubcatalog.loads(catalog["KEYPOINTS2D"])
Keypoints2DSubcatalog(
  (is_tracking): True,
  (keypoints): [...],
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty Keypoints2DSubcatalog and then add the attributes.

```
>>> from tensorbay.label import CategoryInfo, AttributeInfo, KeypointsInfo
>>> from tensorbay.utility import NameList
>>> categories = NameList()
>>> categories.append(CategoryInfo("a"))
>>> attributes = NameList()
>>> attributes.append(AttributeInfo("gender", enum=["female", "male"]))
>>> keypoints2d_subcatalog = Keypoints2DSubcatalog()
>>> keypoints2d_subcatalog.is_tracking = True
>>> keypoints2d_subcatalog.categories = categories
>>> keypoints2d_subcatalog.attributes = attributes
>>> keypoints2d_subcatalog.add_keypoints(
...     2,
...     names=["L_shoulder", "R_Shoulder"],
...     skeleton=[(0,1)],
...     visible="BINARY",
...     parent_categories="shoulder",
...     description="12345",
... )
>>> keypoints2d_subcatalog
Keypoints2DSubcatalog(
  (is_tracking): True,
  (keypoints): [...],
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

property keypoints: List[*tensorbay.label.supports.KeypointsInfo*]

Return the KeypointsInfo of the Subcatalog.

Returns A list of *KeypointsInfo*.

Examples

```
>>> keypoints2d_subcatalog = Keypoints2DSubcatalog()
>>> keypoints2d_subcatalog.add_keypoints(2)
>>> keypoints2d_subcatalog.keypoints
[KeypointsInfo(
  (number): 2
)]
```

add_keypoints(*number*, *, *names=None*, *skeleton=None*, *visible=None*, *parent_categories=None*, *description=''*)

Add a type of keypoints to the subcatalog.

Parameters

- **number** (*int*) – The number of keypoints.
- **names** (*Optional[Iterable[str]]*) – All the names of keypoints.
- **skeleton** (*Optional[Iterable[Iterable[int]]]*) – The skeleton of the keypoints indicating which keypoint should connect with another.
- **visible** (*Optional[str]*) – The visible type of the keypoints, can only be 'BINARY' or 'TERNARY'. It determines the range of the *Keypoint2D.v*.

- **parent_categories** (*Union[None, str, Iterable[str]]*) – The parent categories of the keypoints.
- **description** (*str*) – The description of keypoints.

Return type None

Examples

```
>>> keypoints2d_subcatalog = Keypoints2DSubcatalog()
>>> keypoints2d_subcatalog.add_keypoints(
...     2,
...     names=["L_shoulder", "R_Shoulder"],
...     skeleton=[(0,1)],
...     visible="BINARY",
...     parent_categories="shoulder",
...     description="12345",
... )
>>> keypoints2d_subcatalog.keypoints
[KeypointsInfo(
  (number): 2,
  (names): [...],
  (skeleton): [...],
  (visible): 'BINARY',
  (parent_categories): [...]
)]
```

dumps()

Dumps all the information of the keypoints into a dict.

Returns A dict containing all the information of this Keypoints2DSubcatalog.

Return type Dict[str, Any]

Examples

```
>>> # keypoints2d_subcatalog is the instance initialized above.
>>> keypoints2d_subcatalog.dumps()
{
    'isTracking': True,
    'categories': [{'name': 'a'}],
    'attributes': [{'name': 'gender', 'enum': ['female', 'male']}],
    'keypoints': [
        {
            'number': 2,
            'names': ['L_shoulder', 'R_Shoulder'],
            'skeleton': [(0, 1)],
        }
    ]
}
```

class tensorbay.label.label_keypoints.**LabeledKeypoints2D**(*keypoints=None*, ***, *category=None*, *attributes=None*, *instance=None*)

Bases: tensorbay.label.basic._LabelBase, *tensorbay.geometry.keypoint.Keypoints2D*

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, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

instance

The instance id of the label.

Type str

Examples

```
>>> LabeledKeypoints2D(
...     [(1, 2), (2, 3)],
...     category="example",
...     attributes={"key": "value"},
...     instance="123",
... )
LabeledKeypoints2D [
  Keypoint2D(1, 2),
  Keypoint2D(2, 3)
](
  (category): 'example',
  (attributes): {...},
  (instance): '123'
)
```

classmethod loads(*contents*)

Loads a LabeledKeypoints2D from a dict containing the information of the label.

Parameters **contents** (*Dict[str, Any]*) – A dict containing the information of the 2D keypoints label.

Returns The loaded *LabeledKeypoints2D* object.

Return type tensorbay.label.label_keypoints._T

Examples

```
>>> contents = {
...     "keypoints2d": [
...         {"x": 1, "y": 1, "v": 2},
...         {"x": 2, "y": 2, "v": 2},
...     ],
...     "category": "example",
...     "attributes": {"key": "value"},
...     "instance": "12345",
... }
>>> LabeledKeypoints2D.loads(contents)
LabeledKeypoints2D [
  Keypoint2D(1, 1, 2),
  Keypoint2D(2, 2, 2)
](
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

dumps()

Dumps the current 2D keypoints label into a dict.

Returns A dict containing all the information of the 2D keypoints label.

Return type Dict[str, Any]

Examples

```
>>> labeledkeypoints2d = LabeledKeypoints2D(
...     [(1, 1, 2), (2, 2, 2)],
...     category="example",
...     attributes={"key": "value"},
...     instance="123",
... )
>>> labeledkeypoints2d.dumps()
{
    'category': 'example',
    'attributes': {'key': 'value'},
    'instance': '123',
    'keypoints2d': [{'x': 1, 'y': 1, 'v': 2}, {'x': 2, 'y': 2, 'v': 2}],
}
```

tensorbay.label.label_mask

Mask related classes.

class tensorbay.label.label_mask.**SemanticMaskSubcatalog**(*description=''*)

Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.MaskCategoriesMixin*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for semantic mask type of labels.

Parameters **description** (*str*) –

Return type None

description

The description of the entire semantic mask subcatalog.

Type str

categories

All the possible categories in the corresponding dataset stored in a *NameList* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.supports.MaskCategoryInfo*]

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 *NameList* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking

Whether the Subcatalog contains tracking information.

Examples

Initialization Method 1: Init from SemanticMaskSubcatalog.loads() method.

```
>>> catalog = {
...     "SEMANTIC_MASK": {
...         "categories": [
...             {'name': 'cat', "categoryId": 1},
...             {'name': 'dog', "categoryId": 2}
...         ],
...         "attributes": [{'name': 'occluded', 'type': 'boolean'}],
...     }
... }
>>> SemanticMaskSubcatalog.loads(catalog["SEMANTIC_MASK"])
SemanticMaskSubcatalog(
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty SemanticMaskSubcatalog and then add the attributes.

```
>>> semantic_mask_subcatalog = SemanticMaskSubcatalog()
>>> semantic_mask_subcatalog.add_category("cat", 1)
>>> semantic_mask_subcatalog.add_category("dog", 2)
>>> semantic_mask_subcatalog.add_attribute("occluded", type_="boolean")
>>> semantic_mask_subcatalog
SemanticMaskSubcatalog(
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class tensorbay.label.label_mask.**InstanceMaskSubcatalog**(*description=''*)

Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.MaskCategoriesMixin*, *tensorbay.label.supports.IsTrackingMixin*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for instance mask type of labels.

Parameters **description** (*str*) –

Return type None

description

The description of the entire instance mask subcatalog.

Type str

categories

All the possible categories in the corresponding dataset stored in a *NameList* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.supports.MaskCategoryInfo*]

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 *NameList* with the attribute names as keys and the `AttributeInfo` as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking

Whether the Subcatalog contains tracking information.

Type bool

Examples

Initialization Method 1: Init from `InstanceMaskSubcatalog.loads()` method.

```
>>> catalog = {
...     "INSTANCE_MASK": {
...         "categories": [
...             {'name': 'background', "categoryId": 0}
...         ],
...         "attributes": [{'name': 'occluded', 'type': 'boolean'}],
...     }
```

(continues on next page)

(continued from previous page)

```
... }
>>> InstanceMaskSubcatalog.loads(catalog["INSTANCE_MASK"])
InstanceMaskSubcatalog(
  (is_tracking): False,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty InstanceMaskSubcatalog and then add the attributes.

```
>>> instance_mask_subcatalog = InstanceMaskSubcatalog()
>>> instance_mask_subcatalog.add_category("background", 0)
>>> instance_mask_subcatalog.add_attribute("occluded", type_="boolean")
>>> instance_mask_subcatalog
InstanceMaskSubcatalog(
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class tensorbay.label.label_mask.**PanopticMaskSubcatalog**(*description=''*)

Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.MaskCategoriesMixin*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for panoptic mask type of labels.

Parameters **description** (*str*) –

Return type None

description

The description of the entire panoptic mask subcatalog.

Type str

categories

All the possible categories in the corresponding dataset stored in a *NameList* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.supports.MaskCategoryInfo*]

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 *NameList* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking

Whether the Subcatalog contains tracking information.

Examples

Initialization Method 1: Init from `PanopticMaskSubcatalog.loads()` method.

```
>>> catalog = {
...     "PANOPTIC_MASK": {
...         "categories": [
...             {'name': 'cat', "categoryId": 1},
...             {'name': 'dog', "categoryId": 2}
...         ],
...         "attributes": [{'name': 'occluded', 'type': 'boolean'}],
...     }
... }
>>> PanopticMaskSubcatalog.loads(catalog["PANOPTIC_MASK"])
PanopticMaskSubcatalog(
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty PanopticMaskSubcatalog and then add the attributes.

```
>>> panoptic_mask_subcatalog = PanopticMaskSubcatalog()
>>> panoptic_mask_subcatalog.add_category("cat", 1)
>>> panoptic_mask_subcatalog.add_category("dog", 2)
>>> panoptic_mask_subcatalog.add_attribute("occluded", type_="boolean")
>>> panoptic_mask_subcatalog
PanopticMaskSubcatalog(
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class `tensorbay.label.label_mask.`**`SemanticMaskBase`**

Bases: *`tensorbay.utility.repr.ReprMixin`*

SemanticMaskBase is a base class for the semantic mask label.

`all_attributes`

The dict of the attributes in this mask, which key is the category id, and the value is the corresponding attributes.

Type Dict[int, Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]]

class `tensorbay.label.label_mask.`**`InstanceMaskBase`**

Bases: *`tensorbay.utility.repr.ReprMixin`*

InstanceMaskBase is a base class for the instance mask label.

`all_attributes`

The dict of the attributes in this mask, which key is the instance id, and the value is the corresponding attributes.

Type Dict[int, Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]]

class `tensorbay.label.label_mask.`**`PanopticMaskBase`**

Bases: *`tensorbay.utility.repr.ReprMixin`*

PanopticMaskBase is a base class for the panoptic mask label.

Return type None

all_attributes
The dict of the attributes in this mask, which key is the instance id, and the value is the corresponding attributes.

Type Dict[int, Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]]

all_category_ids
The dict of the category id in this mask, which key is the instance id, and the value is the corresponding category id.

class tensorbay.label.label_mask.**SemanticMask**(*local_path*)
Bases: *tensorbay.label.label_mask.SemanticMaskBase*, *tensorbay.utility.file.FileMixin*

SemanticMask is a class for the local semantic mask label.

Parameters **local_path** (*str*) –

Return type None

all_attributes
The dict of the attributes in this mask, which key is the category id, and the value is the corresponding attributes.

Type Dict[int, Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]]

get_callback_body()
Get the callback request body for uploading.

Returns

The callback request body, which looks like:

```
{
    "checksum": <str>,
    "fileSize": <int>,
    "info": [
        {
            "categoryId": 0,
            "attributes": {
                "occluded": True
            }
        },
        {
            "categoryId": 1,
            "attributes": {
                "occluded": False
            }
        }
    ]
}
```

Return type Dict[str, Any]

class tensorbay.label.label_mask.**InstanceMask**(*local_path*)
Bases: *tensorbay.label.label_mask.InstanceMaskBase*, *tensorbay.utility.file.FileMixin*

InstanceMask is a class for the local instance mask label.

Parameters **local_path** (*str*) –

Return type None

all_attributes
The dict of the attributes in this mask, which key is the instance id, and the value is the corresponding attributes.

Type Dict[int, Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]]

get_callback_body()
Get the callback request body for uploading.

Returns

The callback request body, which looks like:

```
{
    "checksum": <str>,
    "fileSize": <int>,
    "info": [
        {
            "instanceId": 0,
            "attributes": {
                "occluded": True
            }
        },
        {
            "instanceId": 1,
            "attributes": {
                "occluded": False
            }
        }
    ]
}
```

Return type Dict[str, Any]

class tensorbay.label.label_mask.PanopticMask(*local_path*)
Bases: *tensorbay.label.label_mask.PanopticMaskBase*, *tensorbay.utility.file.FileMixin*

PanopticMask is a class for the local panoptic mask label.

Parameters **local_path** (*str*) –

Return type None

all_attributes
The dict of the attributes in this mask, which key is the instance id, and the value is the corresponding attributes.

Type Dict[int, Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]]

all_category_ids
The dict of the category id in this mask, which key is the instance id, and the value is the corresponding category id.

get_callback_body()
Get the callback request body for uploading.

Returns

The callback request body, which looks like:

```
{
    "checksum": <str>,
    "fileSize": <int>,
    "info": [
        {
            "instanceId": 0,
            "categoryId": 100,
            "attributes": {
                "occluded": True
            }
        },
        {
            "instanceId": 1,
            "categoryId": 101,
            "attributes": {
                "occluded": False
            }
        }
    ]
}
```

Return type Dict[str, Any]

class tensorbay.label.label_mask.**RemoteSemanticMask**(*remote_path*, *, *url=None*, *cache_path=''*)

Bases: *tensorbay.label.label_mask.SemanticMaskBase*, *tensorbay.utility.file.RemoteFileMixin*

RemoteSemanticMask is a class for the remote semantic mask label.

Parameters

- **remote_path** (*str*) –
- **url** (*Optional[tensorbay.utility.file.URL]*) –
- **cache_path** (*str*) –

Return type None

all_attributes

The dict of the attributes in this mask, which key is the category id, and the value is the corresponding attributes.

Type Dict[int, Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]]

classmethod from_response_body(*body*)

Loads a *RemoteSemanticMask* object from a response body.

Parameters **body** (*Dict[str, Any]*) – The response body which contains the information of a remote semantic mask, whose format should be like:

```
{
    "remotePath": <str>,
    "info": [
        {
            "categoryId": 0,
            "attributes": {
                "occluded": True
```

(continues on next page)

(continued from previous page)

```
                }
            },
            {
                "categoryId": 1,
                "attributes": {
                    "occluded": False
                }
            }
        ]
}
```

Returns The loaded *RemoteSemanticMask* object.

Return type tensorbay.label.label_mask._T

class tensorbay.label.label_mask.**RemoteInstanceMask**(*remote_path*, *, *url=None*, *cache_path=''*)

Bases: *tensorbay.label.label_mask.InstanceMaskBase*, *tensorbay.utility.file.RemoteFileMixin*

RemoteInstanceMask is a class for the remote instance mask label.

Parameters

- **remote_path** (*str*) –
- **url** (*Optional[tensorbay.utility.file.URL]*) –
- **cache_path** (*str*) –

Return type None

all_attributes

The dict of the attributes in this mask, which key is the instance id, and the value is the corresponding attributes.

Type Dict[int, Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]]

classmethod from_response_body(*body*)

Loads a *RemoteInstanceMask* object from a response body.

Parameters **body** (*Dict[str, Any]*) – The response body which contains the information of a remote instance mask, whose format should be like:

```
{
    "remotePath": <str>,
    "info": [
        {
            "instanceId": 0,
            "attributes": {
                "occluded": True
            }
        },
        {
            "instanceId": 1,
            "attributes": {
                "occluded": False
            }
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

Returns The loaded *RemoteInstanceMask* object.

Return type tensorbay.label.label_mask._T

class tensorbay.label.label_mask.**RemotePanopticMask**(*remote_path*, *, *url=None*)

Bases: *tensorbay.label.label_mask.PanopticMaskBase*, *tensorbay.utility.file.RemoteFileMixin*

RemotePanoticMask is a class for the remote panotic mask label.

Parameters

- **remote_path** (*str*) –
- **url** (*Optional[tensorbay.utility.file.URL]*) –

Return type None

all_attributes

The dict of the attributes in this mask, which key is the instance id, and the value is the corresponding attributes.

Type Dict[int, Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]]

classmethod from_response_body(*body*)

Loads a *RemotePanopticMask* object from a response body.

Parameters **body** (*Dict[str, Any]*) – The response body which contains the information of a remote panoptic mask, whose format should be like:

```
{
    "remotePath": <str>,
    "info": [
        {
            "instanceId": 0,
            "categoryId": 100,
            "attributes": {
                "occluded": True
            }
        },
        {
            "instanceId": 1,
            "categoryId": 101,
            "attributes": {
                "occluded": False
            }
        }
    ]
}
```

Returns The loaded *RemotePanopticMask* object.

Return type tensorbay.label.label_mask._T

tensorbay.label.label_polygon

LabeledPolygon, PolygonSubcatalog.

PolygonSubcatalog defines the subcatalog for polygon type of labels.

LabeledPolygon is the polygon type of label, which is often used for CV tasks such as semantic segmentation.

class tensorbay.label.label_polygon.**PolygonSubcatalog**(*is_tracking=False*)

Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.IsTrackingMixin*, *tensorbay.label.supports.CategoriesMixin*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for polygon type of labels.

Parameters is_tracking (*bool*) – A boolean value indicates whether the corresponding subcatalog contains tracking information.

Return type None

description

The description of the entire polygon subcatalog.

Type str

categories

All the possible categories in the corresponding dataset stored in a *NameList* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameList*[*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 *NameList* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking

Whether the Subcatalog contains tracking information.

Type bool

Examples

Initialization Method 1: Init from PolygonSubcatalog.loads() method.

```
>>> catalog = {
...     "POLYGON": {
...         "isTracking": True,
...         "categories": [{"name": "0"}, {"name": "1"}],
...         "attributes": [{"name": "gender", "enum": ["male", "female"]}],
...     }
... }
>>> PolygonSubcatalog.loads(catalog["POLYGON"])
PolygonSubcatalog(
  (is_tracking): True,
  (categories): NameList [...],
```

(continues on next page)

(continued from previous page)

```
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty PolygonSubcatalog and then add the attributes.

```
>>> from tensorbay.utility import NameList
>>> from tensorbay.label import CategoryInfo, AttributeInfo
>>> categories = NameList()
>>> categories.append(CategoryInfo("a"))
>>> attributes = NameList()
>>> attributes.append(AttributeInfo("gender", enum=["female", "male"]))
>>> polygon_subcatalog = PolygonSubcatalog()
>>> polygon_subcatalog.is_tracking = True
>>> polygon_subcatalog.categories = categories
>>> polygon_subcatalog.attributes = attributes
>>> polygon_subcatalog
PolygonSubcatalog(
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class `tensorbay.label.label_polygon.`**`MultiPolygonSubcatalog`**(*is_tracking=False*)

Bases: *`tensorbay.label.basic.SubcatalogBase`*, *`tensorbay.label.supports.IsTrackingMixin`*, *`tensorbay.label.supports.CategoriesMixin`*, *`tensorbay.label.supports.AttributesMixin`*

This class defines the subcatalog for multiple polygon type of labels.

Parameters **`is_tracking`** (*`bool`*) – A boolean value indicates whether the corresponding subcatalog contains tracking information.

Return type None

`description`

The description of the entire multiple polygon subcatalog.

Type str

`categories`

All the possible categories in the corresponding dataset stored in a *`NameList`* with the category names as keys and the *`CategoryInfo`* as values.

Type *tensorbay.utility.name.NameList*[*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 *`NameList`* with the attribute names as keys and the `AttributeInfo` as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

`is_tracking`

Whether the Subcatalog contains tracking information.

Type bool

Examples

Initialization Method 1: Init from `MultiPolygonSubcatalog.loads()` method.

```
>>> catalog = {
...     "MULTI_POLYGON": {
...         "isTracking": True,
...         "categories": [{"name": "0"}, {"name": "1"}],
...         "attributes": [{"name": "gender", "enum": ["male", "female"]}],
...     }
... }
>>> MultiPolygonSubcatalog.loads(catalog["MULTI_POLYGON"])
MultiPolygonSubcatalog(
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty MultiPolygonSubcatalog and then add the attributes.

```
>>> from tensorbay.label import CategoryInfo, AttributeInfo
>>> multi_polygon_subcatalog = MultiPolygonSubcatalog()
>>> multi_polygon_subcatalog.is_tracking = True
>>> multi_polygon_subcatalog.add_category("a")
>>> multi_polygon_subcatalog.add_attribute("gender", enum=["female", "male"])
>>> multi_polygon_subcatalog
MultiPolyline2DSubcatalog(
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class `tensorbay.label.label_polygon.`**`RLESubcatalog`**(*is_tracking=False*)

Bases: *`tensorbay.label.basic.SubcatalogBase`*, *`tensorbay.label.supports.IsTrackingMixin`*, *`tensorbay.label.supports.CategoriesMixin`*, *`tensorbay.label.supports.AttributesMixin`*

This class defines the subcatalog for rle type of labels.

Parameters **`is_tracking`** (*`bool`*) – A boolean value indicating whether the corresponding subcatalog contains tracking information.

Return type None

`description`

The description of the rle subcatalog.

Type str

`categories`

All the possible categories in the corresponding dataset stored in a *`NameList`* with the category names as keys and the *`CategoryInfo`* as values.

Type *tensorbay.utility.name.NameList*[*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 *NameList* with the attribute names as keys and the `AttributeInfo` as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking
Whether the Subcatalog contains tracking information.

Type bool

Examples

Initialization Method 1: Init from `RLESubcatalog.loads()` method.

```
>>> catalog = {
...     "RLE": {
...         "isTracking": True,
...         "categories": [{"name": "0"}, {"name": "1"}],
...         "attributes": [{"name": "gender", "enum": ["male", "female"]}],
...     }
... }
>>> RLESubcatalog.loads(catalog["RLESubcatalog"])
RLESubcatalog(
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty RLESubcatalog and then add the attributes.

```
>>> from tensorbay.label import CategoryInfo, AttributeInfo
>>> rle_subcatalog = RLESubcatalog()
>>> rle_subcatalog.is_tracking = True
>>> rle_subcatalog.add_category("a")
>>> rle_subcatalog.add_attribute("gender", enum=["female", "male"])
>>> rle_subcatalog
RLESubcatalog(
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class `tensorbay.label.label_polygon.`**LabeledPolygon**(*points=None*, ***, *category=None*, *attributes=None*, *instance=None*)
Bases: `tensorbay.label.basic._LabelBase`, *tensorbay.geometry.polygon.Polygon*

This class defines the concept of polygon label.

LabeledPolygon is the 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 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, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

instance
The instance id of the label.

Type str

Examples

```
>>> LabeledPolygon(
...     [(1, 2), (2, 3), (1, 3)],
...     category = "example",
...     attributes = {"key": "value"},
...     instance = "123",
... )
LabeledPolygon [
  Vector2D(1, 2),
  Vector2D(2, 3),
  Vector2D(1, 3)
](
  (category): 'example',
  (attributes): {...},
  (instance): '123'
)
```

classmethod loads(*contents*)
Loads a LabeledPolygon from a dict containing the information of the label.

Parameters contents (*Dict[str, Any]*) – A dict containing the information of the polygon label.

Returns The loaded *LabeledPolygon* object.

Return type tensorbay.label.label_polygon._T

Examples

```
>>> contents = {
...     "polygon": [
...         {"x": 1, "y": 2},
...         {"x": 2, "y": 3},
...         {"x": 1, "y": 3},
...     ],
...     "category": "example",
...     "attributes": {"key": "value"},
...     "instance": "12345",
```

(continues on next page)

(continued from previous page)

```
... }
>>> LabeledPolygon.loads(contents)
LabeledPolygon [
  Vector2D(1, 2),
  Vector2D(2, 3),
  Vector2D(1, 3)
](
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

dumps()

Dumps the current polygon label into a dict.

Returns A dict containing all the information of the polygon label.

Return type Dict[str, Any]

Examples

```
>>> labeledpolygon = LabeledPolygon(
...     [(1, 2), (2, 3), (1, 3)],
...     category = "example",
...     attributes = {"key": "value"},
...     instance = "123",
... )
>>> labeledpolygon.dumps()
{
    'category': 'example',
    'attributes': {'key': 'value'},
    'instance': '123',
    'polygon': [{'x': 1, 'y': 2}, {'x': 2, 'y': 3}, {'x': 1, 'y': 3}],
}
```

class tensorbay.label.label_polygon.**LabeledMultiPolygon**(*polygons=None*, ***, *category=None*, *attributes=None*, *instance=None*)

Bases: tensorbay.label.basic._LabelBase, *tensorbay.geometry.polygon.MultiPolygon*

This class defines the concept of multiple polygon label.

LabeledMultiPolygon is the multipolygon type of label, which is often used for CV tasks such as semantic segmentation.

Parameters

- **points** – A list of 2D points representing the vertices of the 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, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

instance
The instance id of the label.

Type str

Examples

```
>>> LabeledMultiPolygon(
...     [[(1.0, 2.0), (2.0, 3.0), (1.0, 3.0)], [(1.0, 4.0), (2.0, 3.0), (1.0, 8.
↪0)]],
...     category = "example",
...     attributes = {"key": "value"},
...     instance = "12345",
... )
LabeledMultiPolygon [
    Polygon [...],
    Polygon [...]
    ](
      (category): 'example',
      (attributes): {...},
      (instance): '12345'
    )
```

classmethod loads(*contents*)
Loads a LabeledMultiPolygon from a list of dict containing the information of the label.

Parameters contents (*Dict[str, Any]*) – A dict containing the information of the multi-polygon label.

Returns The loaded *LabeledMultiPolygon* object.

Return type tensorbay.label.label_polygon._T

Examples

```
>>> contents = {
...     "multiPolygon": [
...         [
...             {"x": 1.0, "y": 2.0},
...             {"x": 2.0, "y": 3.0},
...             {"x": 1.0, "y": 3.0},
...        ],
...         [{"x": 1.0, "y": 4.0}, {"x": 2.0, "y": 3.0}, {"x": 1.0, "y": 8.0}],
...     ],
...     "category": "example",
...     "attributes": {"key": "value"},
...     "instance": "12345",
... }
```

(continues on next page)

(continued from previous page)

```
>>> LabeledMultiPolygon.loads(contents)
LabeledMultiPolygon [
  Polygon [...],
  Polygon [...]
](
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

dumps()

Dumps the current multipolygon label into a dict.

Returns A dict containing all the information of the multipolygon label.

Return type Dict[str, Any]

Examples

```
>>> labeledmultipolygon = LabeledMultiPolygon(
...     [[(1, 2), (2, 3), (1, 3)],[(1, 2), (2, 3), (1, 3)]],
...     category = "example",
...     attributes = {"key": "value"},
...     instance = "123",
... )
>>> labeledmultipolygon.dumps()
{
    'category': 'example',
    'attributes': {'key': 'value'},
    'instance': '123',
    'multiPolygon': [
        [{'x': 1, 'y': 2}, {'x': 2, 'y': 3}, {'x': 1, 'y': 3}],
        [{"x": 1.0, "y": 4.0}, {"x": 2.0, "y": 3.0}, {"x": 1.0, "y": 8.0}]
    ]
}
```

class tensorbay.label.label_polygon.**LabeledRLE**(*rle=None*, ***, *category=None*, *attributes=None*, *instance=None*)

Bases: `tensorbay.label.basic._LabelBase`, *tensorbay.geometry.polygon.RLE*

This class defines the concept of rle label.

LabeledRLE is the rle type of label, which is often used for CV tasks such as semantic segmentation.

Parameters

- **rle** – A rle format mask.
- **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, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

instance
The instance id of the label.

Type str

Examples

```
>>> LabeledRLE(
...     [272, 2, 4, 4, 2, 9],
...     category = "example",
...     attributes = {"key": "value"},
...     instance = "12345",
... )
LabeledRLE [
  272,
  2,
  ...
](
    (category): 'example',
    (attributes): {...},
    (instance): '12345'
)
```

classmethod loads(*contents*)
Loads a LabeledRLE from a dict containing the information of the label.

Parameters **contents** (*Dict[str, Any]*) – A dict containing the information of the rle label.

Returns The loaded *LabeledRLE* object.

Return type tensorbay.label.label_polygon._T

Examples

```
>>> contents = {
...     "rle": [272, 2, 4, 4, 2, 9],
...     "category": "example",
...     "attributes": {"key": "value"},
...     "instance": "12345",
... }
>>> LabeledRLE.loads(contents)
LabeledRLE [
  272,
  2,
  ...
](
  (category): 'example',
  (attributes): {...},
```

(continues on next page)

(continued from previous page)

```
  (instance): '12345'
)
```

dumps()

Dumps the current rle label into a dict.

Returns A dict containing all the information of the rle label.

Return type Dict[str, Any]

Examples

```
>>> labeled_rle = LabeledRLE(
...     [272, 2, 4, 4, 2, 9],
...     category = "example",
...     attributes = {"key": "value"},
...     instance = "123",
... )
>>> labeled_rle.dumps()
{
    'category': 'example',
    'attributes': {'key': 'value'},
    'instance': '123',
    'rle': [272, 2, 4, 4, 2, 9]
}
```

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.**Polyline2DSubcatalog**(*is_tracking=False*, *is_beizer_curve=False*)

Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.IsTrackingMixin*, *tensorbay.label.supports.CategoriesMixin*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for 2D polyline type of labels.

Parameters

- **is_tracking** (*bool*) – A boolean value indicates whether the corresponding subcatalog contains tracking information.
- **is_beizer_curve** (*bool*) – A boolean value indicates whether the corresponding subcatalog contains beizer curve information.

Return type None

description

The description of the entire 2D polyline subcatalog.

Type str

categories
All the possible categories in the corresponding dataset stored in a `NameList` with the category names as keys and the `CategoryInfo` as values.

Type *tensorbay.utility.name.NameList*[*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 `NameList` with the attribute names as keys and the `AttributeInfo` as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking
Whether the Subcatalog contains tracking information.

Type bool

is_beizer_curve
Whether the Subcatalog contains beizer curve information.

Type bool

Examples

Initialization Method 1: Init from `Polyline2DSubcatalog.loads()` method.

```
>>> catalog = {
...     "POLYLINE2D": {
...         "isTracking": True,
...         "isBeizerCurve": True,
...         "categories": [{"name": "0"}, {"name": "1"}],
...         "attributes": [{"name": "gender", "enum": ["male", "female"]}],
...     }
... }
>>> Polyline2DSubcatalog.loads(catalog["POLYLINE2D"])
Polyline2DSubcatalog(
  (is_beizer_curve): True,
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty Polyline2DSubcatalog and then add the attributes.

```
>>> from tensorbay.label import CategoryInfo, AttributeInfo
>>> from tensorbay.utility import NameList
>>> categories = NameList()
>>> categories.append(CategoryInfo("a"))
>>> attributes = NameList()
>>> attributes.append(AttributeInfo("gender", enum=["female", "male"]))
>>> polyline2d_subcatalog = Polyline2DSubcatalog()
>>> polyline2d_subcatalog.is_tracking = True
```

(continues on next page)

(continued from previous page)

```
>>> polyline2d_subcatalog.is_beizer_curve = True
>>> polyline2d_subcatalog.categories = categories
>>> polyline2d_subcatalog.attributes = attributes
>>> polyline2d_subcatalog
Polyline2DSubcatalog(
  (is_beizer_curve): True,
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class tensorbay.label.label_polyline.**LabeledPolyline2D**(*points=None*, ***, *category=None*, *attributes=None*, *instance=None*, *beizer_point_types=None*)

Bases: `tensorbay.label.basic._LabelBase`, *tensorbay.geometry.polyline.Polyline2D*

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.
- **beizer_point_types** – The beizer point types of the label.

category

The category of the label.

Type str

attributes

The attributes of the label.

Type Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

instance

The instance id of the label.

Type str

beizer_point_types

The beizer point types of the label.

Type str

Examples

```
>>> LabeledPolyline2D(
...     [(1, 2), (2, 4), (2, 1)],
...     category="example",
...     attributes={"key": "value"},
...     instance="123",
...     beizer_point_types="LLL",
... )
LabeledPolyline2D [
  Vector2D(1, 2),
  Vector2D(2, 4),
  Vector2D(2, 1)
](
  (beizer_point_types): 'LLL',
  (category): 'example',
  (attributes): {...},
  (instance): '123'
)
```

classmethod loads(*contents*)
Loads a LabeledPolyline2D from a dict containing the information of the label.

Parameters **contents** (*Dict[str, Any]*) – A dict containing the information of the 2D polyline label.

Returns The loaded *LabeledPolyline2D* object.

Return type tensorbay.label.label_polyline._T

Examples

```
>>> contents = {
...     "polyline2d": [{'x': 1, 'y': 2}, {'x': 2, 'y': 4}, {'x': 2, 'y': 1}],
...     "category": "example",
...     "attributes": {"key": "value"},
...     "instance": "12345",
...     "beizer_point_types": "LLL",
... }
>>> LabeledPolyline2D.loads(contents)
LabeledPolyline2D [
  Vector2D(1, 2),
  Vector2D(2, 4),
  Vector2D(2, 1)
](
  (beizer_point_types): 'LLL',
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

dumps()
Dumps the current 2D polyline label into a dict.

Returns A dict containing all the information of the 2D polyline label.

Return type Dict[str, Any]

Examples

```
>>> labeledpolyline2d = LabeledPolyline2D(
...     [(1, 2), (2, 4), (2, 1)],
...     category="example",
...     attributes={"key": "value"},
...     instance="123",
...     beizer_point_types="LLL",
... )
>>> labeledpolyline2d.dumps()
{
    'category': 'example',
    'attributes': {'key': 'value'},
    'instance': '123',
    'polyline2d': [{'x': 1, 'y': 2}, {'x': 2, 'y': 4}, {'x': 2, 'y': 1}],
    'beizerPointTypes': 'LLL',
}
```

class tensorbay.label.label_polyline.**MultiPolyline2DSubcatalog**(*is_tracking=False*)

Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.IsTrackingMixin*, *tensorbay.label.supports.CategoriesMixin*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for 2D multiple polyline type of labels.

Parameters **is_tracking** (*bool*) – A boolean value indicates whether the corresponding subcatalog contains tracking information.

Return type None

description

The description of the entire 2D multiple polyline subcatalog.

Type str

categories

All the possible categories in the corresponding dataset stored in a `NameList` with the category names as keys and the `CategoryInfo` as values.

Type *tensorbay.utility.name.NameList*[*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 `NameList` with the attribute names as keys and the `AttributeInfo` as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

is_tracking

Whether the Subcatalog contains tracking information.

Type bool

Examples

Initialization Method 1: Init from `MultiPolyline2DSubcatalog.loads()` method.

```
>>> catalog = {
...     "MULTI_POLYLINE2D": {
...         "isTracking": True,
...         "categories": [{"name": "0"}, {"name": "1"}],
...         "attributes": [{"name": "gender", "enum": ["male", "female"]}],
...     }
... }
>>> MultiPolyline2DSubcatalog.loads(catalog["MULTI_POLYLINE2D"])
MultiPolyline2DSubcatalog(
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

Initialization Method 2: Init an empty MultiPolyline2DSubcatalog and then add the attributes.

```
>>> from tensorbay.label import CategoryInfo, AttributeInfo
>>> multi_polyline2d_subcatalog = MultiPolyline2DSubcatalog()
>>> multi_polyline2d_subcatalog.is_tracking = True
>>> multi_polyline2d_subcatalog.add_category(CategoryInfo("a"))
>>> multi_polyline2d_subcatalog.add_attribute(
    AttributeInfo("gender", enum=["female", "male"]))
>>> multi_polyline2d_subcatalog
MultiPolyline2DSubcatalog(
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class `tensorbay.label.label_polyline.`**`LabeledMultiPolyline2D`**(*polylines=None*, ***, *category=None*, *attributes=None*, *instance=None*)

Bases: `tensorbay.label.basic._LabelBase`, *`tensorbay.geometry.polyline.MultiPolyline2D`*

This class defines the concept of multiPolyline2D label.

`LabeledMultiPolyline2D` is the 2D multiple polyline type of label, which is often used for CV tasks such as lane detection.

Parameters

- **polylines** – A list of polylines.
- **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, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

instance
The instance id of the label.

Type str

Examples

```
>>> LabeledMultiPolyline2D(
...     [[[1, 2], [2, 3]], [[3, 4], [6, 8]]],
...     category="example",
...     attributes={"key": "value"},
...     instance="123",
... )
LabeledPolyline2D [
  Polyline2D [...]
  Polyline2D [...]
](
  (category): 'example',
  (attributes): {...},
  (instance): '123'
)
```

classmethod loads(*contents*)
Loads a LabeledMultiPolyline2D from a dict containing the information of the label.

Parameters **contents** (*Dict[str, Any]*) – A dict containing the information of the 2D polyline label.

Returns The loaded *LabeledMultiPolyline2D* object.

Return type tensorbay.label.label_polyline._T

Examples

```
>>> contents = {
...     "multiPolyline2d": [[{'x': 1, 'y': 1}, {'x': 1, 'y': 2}, {'x': 2, 'y':␣
↪2}],
                            [{'x': 2, 'y': 3}, {'x': 3, 'y': 5}]],
...     "category": "example",
...     "attributes": {"key": "value"},
...     "instance": "12345",
... }
>>> LabeledMultiPolyline2D.loads(contents)
LabeledMultiPolyline2D [
  Polyline2D [...]
  Polyline2D [...]
](
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

dumps()

Dumps the current 2D multiple polyline label into a dict.

Returns A dict containing all the information of the 2D polyline label.

Return type Dict[str, Any]

Examples

```
>>> labeledmultipolyline2d = LabeledMultiPolyline2D(
...     [[[1, 1], [1, 2], [2, 2]], [[2, 3], [3, 5]]],
...     category="example",
...     attributes={"key": "value"},
...     instance="123",
... )
>>> labeledpolyline2d.dumps()
{
    'category': 'example',
    'attributes': {'key': 'value'},
    'instance': '123',
    'polyline2d': [
        [{'x': 1, 'y': 1}, {'x': 1, 'y': 2}, {'x': 2, 'y': 2}],
        [{'x': 2, 'y': 3}, {'x': 3, 'y': 5}],
}
```

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.**SentenceSubcatalog**(*is_sample=False*, *sample_rate=None*, *lexicon=None*)

Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for audio transcripted sentence type of labels.

Parameters

- **is_sample** (*bool*) – A boolen value indicates whether time format is sample related.
- **sample_rate** (*int*) – The number of samples of audio carried per second.
- **lexicon** (*List[List[str]]*) – A list consists all of text and phone.

Return type None

description

The description of the entire sentence subcatalog.

Type str

is_sample
A boolen value indicates whether time format is sample related.

Type bool

sample_rate
The number of samples of audio carried per second.

Type int

lexicon
A list consists all of text and phone.

Type List[List[str]]

attributes
All the possible attributes in the corresponding dataset stored in a *NameList* with the attribute names as keys and the AttributeInfo as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

Raises **TypeError** – When sample_rate is None and is_sample is True.

Parameters

- **is_sample** (*bool*) –
- **sample_rate** (*int*) –
- **lexicon** (*List[List[str]]*) –

Return type None

Examples

Initialization Method 1: Init from SentenceSubcatalog.__init__().

```
>>> SentenceSubcatalog(True, 16000, [["mean", "m", "iy", "n"]])
SentenceSubcatalog(
  (is_sample): True,
  (sample_rate): 16000,
  (lexicon): [...]
)
```

Initialization Method 2: Init from SentenceSubcatalog.loads() method.

```
>>> contents = {
...     "isSample": True,
...     "sampleRate": 16000,
...     "lexicon": [["mean", "m", "iy", "n"]],
...     "attributes": [{"name": "gender", "enum": ["male", "female"]}],
... }
>>> SentenceSubcatalog.loads(contents)
SentenceSubcatalog(
  (is_sample): True,
  (sample_rate): 16000,
  (attributes): NameList [...],
  (lexicon): [...]
)
```

dumps()

Dumps the information of this SentenceSubcatalog into a dict.

Returns A dict containing all information of this SentenceSubcatalog.

Return type Dict[str, Any]

Examples

```
>>> sentence_subcatalog = SentenceSubcatalog(True, 16000, [["mean", "m", "iy",
↪"n"]])
>>> sentence_subcatalog.dumps()
{'isSample': True, 'sampleRate': 16000, 'lexicon': [['mean', 'm', 'iy', 'n']]}
```

append_lexicon(*lexemes*)

Add lexemes to lexicon.

Parameters **lexemes** (*List[str]*) – A list consists of text and phone.

Return type None

Examples

```
>>> sentence_subcatalog = SentenceSubcatalog(True, 16000, [["mean", "m", "iy",
↪"n"]])
>>> sentence_subcatalog.append_lexicon(["example"])
>>> sentence_subcatalog.lexicon
[['mean', 'm', 'iy', 'n'], ['example']]
```

class tensorbay.label.label_sentence.**Word**(*text*, *begin=None*, *end=None*)

Bases: *tensorbay.utility.repr.ReprMixin*, *tensorbay.utility.attr.AttrsMixin*

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** (*str*) – The content of the word.
- **begin** (*float*) – The begin time of the word in the audio.
- **end** (*float*) – The end time of the word in the audio.

text

The content of the word.

Type str

begin

The begin time of the word in the audio.

Type float

end

The end time of the word in the audio.

Type float

Examples

```
>>> Word(text="example", begin=1, end=2)
Word(
  (text): 'example',
  (begin): 1,
  (end): 2
)
```

classmethod loads(*contents*)

Loads a Word from a dict containing the information of the word.

Parameters **contents** (*Dict[str, Union[str, float]]*) – A dict containing the information of the word

Returns The loaded *Word* object.

Return type tensorbay.label.label_sentence._T

Examples

```
>>> contents = {"text": "Hello, World", "begin": 1, "end": 2}
>>> Word.loads(contents)
Word(
  (text): 'Hello, World',
  (begin): 1,
  (end): 2
)
```

dumps()

Dumps the current word into a dict.

Returns A dict containing all the information of the word

Return type Dict[str, Union[str, float]]

Examples

```
>>> word = Word(text="example", begin=1, end=2)
>>> word.dumps()
{'text': 'example', 'begin': 1, 'end': 2}
```

class tensorbay.label.label_sentence.**LabeledSentence**(*sentence=None*, *spell=None*, *phone=None*, ***, *attributes=None*)

Bases: `tensorbay.label.basic._LabelBase`

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** (*List[tensorbay.label.label_sentence.Word]*) – A list of sentence.
- **spell** (*List[tensorbay.label.label_sentence.Word]*) – A list of spell, only exists in Chinese language.

- **phone** (*List[*`tensorbay.label.label_sentence.Word`*]*) – A list of phone.
- **attributes** (*Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]*) – The attributes of the label.

sentence
The transcripted sentence.

Type List[*tensorbay.label.label_sentence.Word*]

spell
The spell within the sentence, only exists in Chinese language.

Type List[*tensorbay.label.label_sentence.Word*]

phone
The phone of the sentence label.

Type List[*tensorbay.label.label_sentence.Word*]

attributes
The attributes of the label.

Type Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

Examples

```
>>> sentence = [Word(text="qi1shi2", begin=1, end=2)]
>>> spell = [Word(text="qi1", begin=1, end=2)]
>>> phone = [Word(text="q", begin=1, end=2)]
>>> LabeledSentence(
...     sentence,
...     spell,
...     phone,
...     attributes={"key": "value"},
... )
LabeledSentence(
  (sentence): [
    Word(
      (text): 'qi1shi2',
      (begin): 1,
      (end): 2
    )
  ],
  (spell): [
    Word(
      (text): 'qi1',
      (begin): 1,
      (end): 2
    )
  ],
  (phone): [
    Word(
      (text): 'q',
      (begin): 1,
      (end): 2
    )
```

(continues on next page)

(continued from previous page)

```
  ],
  (attributes): {
    'key': 'value'
  }
)
```

classmethod loads(*contents*)

Loads a LabeledSentence from a dict containing the information of the label.

Parameters **contents** (*Dict[str, Any]*) – A dict containing the information of the sentence label.

Returns The loaded *LabeledSentence* object.

Return type tensorbay.label.label_sentence._T

Examples

```
>>> contents = {
...     "sentence": [{"text": "qi1shi2", "begin": 1, "end": 2}],
...     "spell": [{"text": "qi1", "begin": 1, "end": 2}],
...     "phone": [{"text": "q", "begin": 1, "end": 2}],
...     "attributes": {"key": "value"},
... }
>>> LabeledSentence.loads(contents)
LabeledSentence(
  (sentence): [
    Word(
      (text): 'qi1shi2',
      (begin): 1,
      (end): 2
    )
  ],
  (spell): [
    Word(
      (text): 'qi1',
      (begin): 1,
      (end): 2
    )
  ],
  (phone): [
    Word(
      (text): 'q',
      (begin): 1,
      (end): 2
    )
  ],
  (attributes): {
    'key': 'value'
  }
)
```

dumps()
Dumps the current label into a dict.

Returns A dict containing all the information of the sentence label.

Return type Dict[str, Any]

Examples

```
>>> sentence = [Word(text="qi1shi2", begin=1, end=2)]
>>> spell = [Word(text="qi1", begin=1, end=2)]
>>> phone = [Word(text="q", begin=1, end=2)]
>>> labeledsentence = LabeledSentence(
...     sentence,
...     spell,
...     phone,
...     attributes={"key": "value"},
... )
>>> labeledsentence.dumps()
{
    'attributes': {'key': 'value'},
    'sentence': [{'text': 'qi1shi2', 'begin': 1, 'end': 2}],
    'spell': [{'text': 'qi1', 'begin': 1, 'end': 2}],
    'phone': [{'text': 'q', 'begin': 1, 'end': 2}]
}
```

tensorbay.label.supports

CatagoryInfo, MaskCategoryInfo, KeypointsInfo and different SubcatalogMixin classes.

`CatagoryInfo` defines a category with the name and description of it.

MaskCategoryInfo defines a category with the name, id and description of it.

KeypointsInfo defines the structure of a set of keypoints.

Table 1.10: 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.**CategoryInfo**(*name*, *description=''*)
Bases: *tensorbay.utility.name.NameMixin*

This class represents the information of a category, including category name and description.

Parameters

- **name** (*str*) – The name of the category.
- **description** (*str*) – The description of the category.

Return type None

name
The name of the category.

description
The description of the category.

Type str

Examples

```
>>> CategoryInfo(name="example", description="This is an example")
CategoryInfo("example")
```

classmethod loads(*contents*)
Loads a CategoryInfo from a dict containing the category.

Parameters **contents** (*Dict[str, str]*) – A dict containing the information of the category.

Returns The loaded *CategoryInfo* object.

Return type tensorbay.label.supports._T

Examples

```
>>> contents = {"name": "example", "description": "This is an exmaple"}
>>> CategoryInfo.loads(contents)
CategoryInfo("example")
```

dumps()
Dumps the CatagoryInfo into a dict.

Returns A dict containing the information in the CategoryInfo.

Return type Dict[str, str]

Examples

```
>>> categoryinfo = CategoryInfo(name="example", description="This is an example
↪")
>>> categoryinfo.dumps()
{'name': 'example', 'description': 'This is an example'}
```

class tensorbay.label.supports.**MaskCategoryInfo**(*name*, *category_id*, *description=''*)
Bases: *tensorbay.label.supports.CategoryInfo*

This class represents the information of a category, including name, id and description.

Parameters

- **name** (*str*) – The name of the category.
- **category_id** (*int*) – The id of the category.
- **description** (*str*) – The description of the category.

Return type None

name
The name of the category.

category_id
The id of the category.

Type int

description
The description of the category.

Type str

Examples

```
>>> MaskCategoryInfo(name="example", category_id=1, description="This is an example
↪")
MaskCategoryInfo("example")(
  (category_id): 1
)
```

class tensorbay.label.supports.**KeypointsInfo**(*number*, ***, *names=None*, *skeleton=None*, *visible=None*, *parent_categories=None*, *description=''*)

Bases: *tensorbay.utility.repr.ReprMixin*, *tensorbay.utility.attr.AttrsMixin*

This class defines the structure of a set of keypoints.

Parameters

- **number** (*int*) – The number of the set of keypoints.
- **names** (*List[str]*) – All the names of the keypoints.
- **skeleton** (*List[Tuple[int, int]]*) – The skeleton of the keypoints indicating which keypoint should connect with another.
- **visible** (*str*) – The visible type of the keypoints, can only be 'BINARY' or 'TERNARY'. It determines the range of the *Keypoint2D.v*.
- **parent_categories** (*List[str]*) – The parent categories of the keypoints.
- **description** (*str*) – The description of the keypoints.

number
The number of the set of keypoints.

names
All the names of the keypoints.

Type List[str]

skeleton
The skeleton of the keypoints indicating which keypoint should connect with another.

Type List[Tuple[int, int]]

visible
The visible type of the keypoints, can only be 'BINARY' or 'TERNARY'. It determines the range of the *Keypoint2D.v*.

Type str

parent_categories
The parent categories of the keypoints.

Type List[str]

description
The description of the keypoints.

Type str

Examples

```
>>> KeypointsInfo(
...     2,
...     names=["L_Shoulder", "R_Shoulder"],
...     skeleton=[(0, 1)],
...     visible="BINARY",
...     parent_categories="people",
...     description="example",
... )
KeypointsInfo(
  (number): 2,
  (names): [...],
  (skeleton): [...],
  (visible): 'BINARY',
  (parent_categories): [...]
)
```

classmethod loads(*contents*)
Loads a KeypointsInfo from a dict containing the information of the keypoints.

Parameters **contents** (*Dict[str, Any]*) – A dict containing all the information of the set of keypoints.

Returns The loaded *KeypointsInfo* object.

Return type tensorbay.label.supports._T

Examples

```
>>> contents = {
...     "number": 2,
...     "names": ["L", "R"],
...     "skeleton": [(0,1)],
...     "visible": "TERNARY",
...     "parentCategories": ["example"],
...     "description": "example",
... }
>>> KeypointsInfo.loads(contents)
KeypointsInfo(
  (number): 2,
  (names): [...],
  (skeleton): [...],
  (visible): 'TERNARY',
  (parent_categories): [...]
)
```

dumps()
Dumps all the keypoint information into a dict.

Returns A dict containing all the information of the keypoint.

Return type Dict[str, Any]

Examples

```
>>> keypointsinfo = KeypointsInfo(
...     2,
...     names=["L_Shoulder", "R_Shoulder"],
...     skeleton=[(0, 1)],
...     visible="BINARY",
...     parent_categories="people",
...     description="example",
... )
>>> keypointsinfo.dumps()
{
    'number': 2,
    'names': ['L_Shoulder', 'R_Shoulder'],
    'skeleton': [(0, 1)],
    'visible': 'BINARY',
    'parentCategories': ['people'],
    'description': 'example',
}
```

class `tensorbay.label.supports.`**`IsTrackingMixin`**(*is_tracking=False*)

Bases: *tensorbay.utility.attr.AttrsMixin*

A mixin class supporting tracking information of a subcatalog.

Parameters **is_tracking** (*bool*) – Whether the Subcatalog contains tracking information.

Return type None

is_tracking

Whether the Subcatalog contains tracking information.

Type bool

class `tensorbay.label.supports.`**`CategoriesMixin`**

Bases: *tensorbay.utility.attr.AttrsMixin*

A mixin class supporting category information of a subcatalog.

categories

All the possible categories in the corresponding dataset stored in a *NameList* with the category names as keys and the *CategoryInfo* as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.supports.CategoryInfo*]

category_delimiter

The delimiter in category values indicating parent-child relationship.

Type str

get_category_to_index()

Return the dict containing the conversion from category to index.

Returns A dict containing the conversion from category to index.

Return type Dict[str, int]

get_index_to_category()

Return the dict containing the conversion from index to category.

Returns A dict containing the conversion from index to category.

Return type Dict[int, str]

add_category(*name*, *description=''*)

Add a category to the Subcatalog.

Parameters

- **name** (*str*) – The name of the category.
- **description** (*str*) – The description of the category.

Return type None

class tensorbay.label.supports.**MaskCategoriesMixin**

Bases: *tensorbay.utility.attr.AttrsMixin*

A mixin class supporting category information of a MaskSubcatalog.

categories

All the possible categories in the corresponding dataset stored in a *NameList* with the category names as keys and the *MaskCategoryInfo* as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.supports.MaskCategoryInfo*]

category_delimiter

The delimiter in category values indicating parent-child relationship.

Type str

get_category_to_index()

Return the dict containing the conversion from category name to category id.

Returns A dict containing the conversion from category name to category id.

Return type Dict[str, int]

get_index_to_category()

Return the dict containing the conversion from category id to category name.

Returns A dict containing the conversion from category id to category name.

Return type Dict[int, str]

add_category(*name*, *category_id*, *description=''*)

Add a category to the Subcatalog.

Parameters

- **name** (*str*) – The name of the category.
- **category_id** (*int*) – The id of the category.
- **description** (*str*) – The description of the category.

Return type None

class tensorbay.label.supports.**AttributesMixin**

Bases: *tensorbay.utility.attr.AttrsMixin*

A mixin class supporting attribute information of a subcatalog.

attributes
All the possible attributes in the corresponding dataset stored in a `NameList` with the attribute names as keys and the `AttributeInfo` as values.

Type *tensorbay.utility.name.NameList*[*tensorbay.label.attributes.AttributeInfo*]

add_attribute(*name*, ***, *type_=''*, *enum=None*, *minimum=None*, *maximum=None*, *items=None*, *parent_categories=None*, *description=''*)
Add an attribute to the Subcatalog.

Parameters

- **name** (*str*) – 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** (*Optional[Iterable[Optional[Union[str, float, bool]]]]*) – All the possible values of an enumeration attribute.
- **minimum** (*Optional[float]*) – The minimum value of number type attribute.
- **maximum** (*Optional[float]*) – The maximum value of number type attribute.
- **items** (*Optional[*`tensorbay.label.attributes.Items`*]*) – The items inside array type attributes.
- **parent_categories** (*Union[None, str, Iterable[str]]*) – The parent categories of the attribute.
- **description** (*str*) – The description of the attributes.
- **type_** (*Union[str, None, Type[Optional[Union[list, bool, int, float, str]]], Iterable[Union[str, None, Type[Optional[Union[list, bool, int, float, str]]]]]]*) –

Return type None

1.24.5 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.**CameraMatrix**(*fx=None*, *fy=None*, *cx=None*, *cy=None*, *skew=0*, ***, *matrix=None*)
Bases: `tensorbay.utility.repr.ReprMixin`, `tensorbay.utility.attr.AttrsMixin`

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

- **fx** (*float*) – The x axis focal length expressed in pixels.
- **fy** (*float*) – The y axis focal length expressed in pixels.
- **cx** (*float*) – The x coordinate of the so called principal point that should be in the center of the image.
- **cy** (*float*) – The y coordinate of the so called principal point that should be in the center of the image.
- **skew** (*float*) – It causes shear distortion in the projected image.
- **matrix** (*Optional[Union[Sequence[Sequence[float]], numpy.ndarray]]*) – A 3x3 Sequence of camera matrix.

Return type None

fx

The x axis focal length expressed in pixels.

Type float

fy

The y axis focal length expressed in pixels.

Type float

cx

The x coordinate of the so called principal point that should be in the center of the image.

Type float

cy

The y coordinate of the so called principal point that should be in the center of the image.

Type float

skew

It causes shear distortion in the projected image.

Type float

Raises TypeError – When only keyword arguments with incorrect keys are provided, or when no arguments are provided.

Parameters

- **fx** (*float*) –
- **fy** (*float*) –
- **cx** (*float*) –
- **cy** (*float*) –
- **skew** (*float*) –
- **matrix** (*Optional[Union[Sequence[Sequence[float]], numpy.ndarray]]*) –

Return type None

Examples

```
>>> matrix = [[1, 3, 3],
...           [0, 2, 4],
...           [0, 0, 1]]
```

Initialazation Method 1: Init from 3x3 sequence array.

```
>>> camera_matrix = CameraMatrix(matrix=matrix)
>>> camera_matrix
CameraMatrix(
    (fx): 1,
    (fy): 2,
    (cx): 3,
    (cy): 4,
    (skew): 3
)
```

Initialazation Method 2: Init from camera calibration parameters, skew is optional.

```
>>> camera_matrix = CameraMatrix(fx=1, fy=2, cx=3, cy=4, skew=3)
>>> camera_matrix
CameraMatrix(
    (fx): 1,
    (fy): 2,
    (cx): 3,
    (cy): 4,
    (skew): 3
)
```

classmethod loads(*contents*)

Loads CameraMatrix from a dict containing the information of the camera matrix.

Parameters **contents** (*Mapping[str, float]*) – A dict containing the information of the camera matrix.

Returns A *CameraMatrix* instance contains the information from the contents dict.

Return type tensorbay.sensor.intrinsics._T

Examples

```
>>> contents = {
...     "fx": 2,
...     "fy": 6,
...     "cx": 4,
...     "cy": 7,
...     "skew": 3
... }
>>> camera_matrix = CameraMatrix.loads(contents)
>>> camera_matrix
CameraMatrix(
    (fx): 2,
    (fy): 6,
```

(continues on next page)

(continued from previous page)

```
    (cx): 4,
    (cy): 7,
    (skew): 3
)
```

dumps()

Dumps the camera matrix into a dict.

Returns A dict containing the information of the camera matrix.

Return type Dict[str, float]

Examples

```
>>> camera_matrix.dumps()
{'fx': 1, 'fy': 2, 'cx': 3, 'cy': 4, 'skew': 3}
```

as_matrix()

Return the camera matrix as a 3x3 numpy array.

Returns A 3x3 numpy array representing the camera matrix.

Return type numpy.ndarray

Examples

```
>>> numpy_array = camera_matrix.as_matrix()
>>> numpy_array
array([[1., 3., 3.],
       [0., 4., 4.],
       [0., 0., 1.]])
```

project(*point*)

Project a point to the pixel coordinates.

Parameters **point** (*Sequence[float]*) – 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.

Return type *tensorbay.geometry.vector.Vector2D*

Examples

Project a point in 2 dimensions

```
>>> camera_matrix.project([1, 2])
Vector2D(12, 19)
```

Project a point in 3 dimensions

```
>>> camera_matrix.project([1, 2, 4])
Vector2D(6.0, 10.0)
```

class tensorbay.sensor.intrinsics.**DistortionCoefficients**(***kwargs*)
Bases: *tensorbay.utility.repr.ReprMixin*, *tensorbay.utility.attr.AttrsMixin*

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, …
- **kwargs** (*float*) –

Raises **TypeError** – When tangential and radial distortion is not provided to initialize class.

Return type None

Examples

```
>>> distortion_coefficients = DistortionCoefficients(p1=1, p2=2, k1=3, k2=4)
>>> distortion_coefficients
DistortionCoefficients(
    (p1): 1,
    (p2): 2,
    (k1): 3,
    (k2): 4
)
```

classmethod loads(*contents*)
Loads DistortionCoefficients from a dict containing the information.

Parameters **contents** (*Mapping[str, float]*) – A dict containig distortion coefficients of a camera.

Returns A *DistortionCoefficients* instance containing information from the contents dict.

Return type tensorbay.sensor.intrinsics._T

Examples

```
>>> contents = {
...     "p1": 1,
...     "p2": 2,
...     "k1": 3,
...     "k2": 4
... }
>>> distortion_coefficients = DistortionCoefficients.loads(contents)
>>> distortion_coefficients
DistortionCoefficients(
    (p1): 1,
    (p2): 2,
    (k1): 3,
```

(continues on next page)

(continued from previous page)

```
    (k2): 4
)
```

dumps()

Dumps the distortion coefficients into a dict.

Returns A dict containing the information of distortion coefficients.

Return type Dict[str, float]

Examples

```
>>> distortion_coefficients.dumps()
{'p1': 1, 'p2': 2, 'k1': 3, 'k2': 4}
```

distort(*point*, *is_fisheye=False*)

Add distortion to a point.

Parameters

- **point** (*Sequence[float]*) – A Sequence containing the coordinates of the point to be distorted.
- **is_fisheye** (*bool*) – 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.

Return type *tensorbay.geometry.vector.Vector2D*

Examples

Distort a point with 2 dimensions

```
>>> distortion_coefficients.distort((1.0, 2.0))
Vector2D(134.0, 253.0)
```

Distort a point with 3 dimensions

```
>>> distortion_coefficients.distort((1.0, 2.0, 3.0))
Vector2D(3.3004115226337447, 4.934156378600823)
```

Distort a point with 2 dimensions, fisheye is True

```
>>> distortion_coefficients.distort((1.0, 2.0), is_fisheye=True)
Vector2D(6.158401093771876, 12.316802187543752)
```

class tensorbay.sensor.intrinsics.**CameraIntrinsics**(*fx=None*, *fy=None*, *cx=None*, *cy=None*, *skew=0*, ***, *camera_matrix=None*, ***kwargs*)

Bases: `tensorbay.utility.repr.ReprMixin`, `tensorbay.utility.attr.AttrsMixin`

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

- **fx** (*Optional[float]*) – The x axis focal length expressed in pixels.
- **fy** (*Optional[float]*) – The y axis focal length expressed in pixels.
- **cx** (*Optional[float]*) – The x coordinate of the so called principal point that should be in the center of the image.
- **cy** (*Optional[float]*) – The y coordinate of the so called principal point that should be in the center of the image.
- **skew** (*float*) – It causes shear distortion in the projected image.
- **camera_matrix** (`tensorbay.sensor.intrinsics.CameraMatrix`) – A 3x3 Sequence of the camera matrix.
- ****kwargs** – Float values to initialize *DistortionCoefficients*.
- **kwargs** (*float*) –

Return type None

camera_matrix

A 3x3 Sequence of the camera matrix.

Type *tensorbay.sensor.intrinsics.CameraMatrix*

distortion_coefficients

It is the deviation from rectilinear projection. It includes

Type *tensorbay.sensor.intrinsics.DistortionCoefficients*

radial distortion and tangential distortion.

Examples

```
>>> matrix = [[1, 3, 3],
...           [0, 2, 4],
...           [0, 0, 1]]
```

Initialization Method 1: Init from 3x3 sequence array.

```
>>> camera_intrinsics = CameraIntrinsics(camera_matrix=matrix, p1=5, k1=6)
>>> camera_intrinsics
CameraIntrinsics(
    (camera_matrix): CameraMatrix(
            (fx): 1,
            (fy): 2,
            (cx): 3,
            (cy): 4,
            (skew): 3
        ),
    (distortion_coefficients): DistortionCoefficients(
            (p1): 5,
            (k1): 6
        )
)
```

Initialization Method 2: Init from camera calibration parameters, skew is optional.

```
>>> camera_intrinsics = CameraIntrinsics(
...     fx=1,
...     fy=2,
...     cx=3,
...     cy=4,
...     p1=5,
...     k1=6,
...     skew=3
... )
>>> camera_intrinsics
CameraIntrinsics(
    (camera_matrix): CameraMatrix(
        (fx): 1,
        (fy): 2,
        (cx): 3,
        (cy): 4,
        (skew): 3
    ),
    (distortion_coefficients): DistortionCoefficients(
        (p1): 5,
        (k1): 6
    )
)
```

classmethod loads(*contents*)

Loads CameraIntrinsics from a dict containing the information.

Parameters **contents** (*Mapping[str, Mapping[str, float]]*) – A dict containig camera matrix and distortion coefficients.

Returns A *CameraIntrinsics* instance containing information from the contents dict.

Return type tensorbay.sensor.intrinsics._T

Examples

```
>>> contents = {
...     "cameraMatrix": {
...         "fx": 1,
...         "fy": 2,
...         "cx": 3,
...         "cy": 4,
...     },
...     "distortionCoefficients": {
...         "p1": 1,
...         "p2": 2,
...         "k1": 3,
...         "k2": 4
...     },
... }
>>> camera_intrinsics = CameraIntrinsics.loads(contents)
>>> camera_intrinsics
CameraIntrinsics(
```

(continues on next page)

(continued from previous page)

```
    (camera_matrix): CameraMatrix(
        (fx): 1,
        (fy): 2,
        (cx): 3,
        (cy): 4,
        (skew): 0
    ),
    (distortion_coefficients): DistortionCoefficients(
        (p1): 1,
        (p2): 2,
        (k1): 3,
        (k2): 4
    )
)
```

dumps()

Dumps the camera intrinsics into a dict.

Returns A dict containing camera intrinsics.

Return type Dict[str, Dict[str, float]]

Examples

```
>>> camera_intrinsics.dumps()
{'cameraMatrix': {'fx': 1, 'fy': 2, 'cx': 3, 'cy': 4, 'skew': 3},
'distortionCoefficients': {'p1': 5, 'k1': 6}}
```

set_camera_matrix(*fx=None*, *fy=None*, *cx=None*, *cy=None*, *skew=0*, ***, *matrix=None*)

Set camera matrix of the camera intrinsics.

Parameters

- **fx** (*Optional[float]*) – The x axis focal length expressed in pixels.
- **fy** (*Optional[float]*) – The y axis focal length expressed in pixels.
- **cx** (*Optional[float]*) – The x coordinate of the so called principal point that should be in the center of the image.
- **cy** (*Optional[float]*) – The y coordinate of the so called principal point that should be in the center of the image.
- **skew** (*float*) – It causes shear distortion in the projected image.
- **matrix** (*Optional[Union[Sequence[Sequence[float]], numpy.ndarray]]*) – Camera matrix in 3x3 sequence.

Return type None

Examples

```
>>> camera_intrinsics.set_camera_matrix(fx=11, fy=12, cx=13, cy=14, skew=15)
>>> camera_intrinsics
CameraIntrinsics(
    (camera_matrix): CameraMatrix(
        (fx): 11,
        (fy): 12,
        (cx): 13,
        (cy): 14,
        (skew): 15
    ),
    (distortion_coefficients): DistortionCoefficients(
        (p1): 1,
        (p2): 2,
        (k1): 3,
        (k2): 4
    )
)
```

set_distortion_coefficients(***kwargs*)

Set distortion coefficients of the camera intrinsics.

Parameters

- ****kwargs** – Contains p1, p2, …, k1, k2, …
- **kwargs** (*float*) –

Return type None

Examples

```
>>> camera_intrinsics.set_distortion_coefficients(p1=11, p2=12, k1=13, k2=14)
>>> camera_intrinsics
CameraIntrinsics(
    (camera_matrix): CameraMatrix(
        (fx): 11,
        (fy): 12,
        (cx): 13,
        (cy): 14,
        (skew): 15
    ),
    (distortion_coefficients): DistortionCoefficients(
        (p1): 11,
        (p2): 12,
        (k1): 13,
        (k2): 14
    )
)
```

project(*point*, *is_fisheye=False*)

Project a point to the pixel coordinates.

If distortion coefficients are provided, distort the point before projection.

Parameters

- **point** (*Sequence[float]*) – A Sequence containing coordinates of the point to be projected.
- **is_fisheye** (*bool*) – Whether the sensor is fisheye camera, default is False.

Returns The coordinates on the pixel plane where the point is projected to.

Return type *tensorbay.geometry.vector.Vector2D*

Examples

Project a point with 2 dimensions.

```
>>> camera_intrinsics.project((1, 2))
Vector2D(137.0, 510.0)
```

Project a point with 3 dimensions.

```
>>> camera_intrinsics.project((1, 2, 3))
Vector2D(6.300411522633745, 13.868312757201647)
```

Project a point with 2 dimensions, fisheye is True

```
>>> camera_intrinsics.project((1, 2), is_fisheye=True)
Vector2D(9.158401093771875, 28.633604375087504)
```

tensorbay.sensor.sensor

SensorType, Sensor, Lidar, Radar, Camera, FisheyeCamera and Sensors.

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.

Sensors represent all the sensors in a *FusionSegment*.

class `tensorbay.sensor.sensor.`**SensorType**(*value*)

Bases: *tensorbay.utility.type.TypeEnum*

SensorType is an enumeration type.

It includes 'LIDAR', 'RADAR', 'CAMERA' and 'FISHEYE_CAMERA'.

Examples

```
>>> SensorType.CAMERA
<SensorType.CAMERA: 'CAMERA'>
>>> SensorType["CAMERA"]
<SensorType.CAMERA: 'CAMERA'>
```

```
>>> SensorType.CAMERA.name
'CAMERA'
>>> SensorType.CAMERA.value
'CAMERA'
```

class `tensorbay.sensor.sensor.`**`Sensor`**(*name*)

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** (*str*) – *Sensor*'s name.

Raises **TypeError** – Can not instantiate abstract class *Sensor*.

Return type tensorbay.sensor.sensor._T

extrinsics

The translation and rotation of the sensor.

Type *tensorbay.geometry.transform.Transform3D*

static loads(*contents*)

Loads a Sensor from a dict containing the sensor information.

Parameters **contents** (*Dict[str, Any]*) – A dict containing name, description and sensor extrinsics.

Returns A *Sensor* instance containing the information from the contents dict.

Return type _Type

Examples

```
>>> contents = {
...     "name": "Lidar1",
...     "type": "LIDAR",
...     "extrinsics": {
...         "translation": {"x": 1.1, "y": 2.2, "z": 3.3},
...         "rotation": {"w": 1.1, "x": 2.2, "y": 3.3, "z": 4.4},
...     },
... }
>>> sensor = Sensor.loads(contents)
>>> sensor
Lidar("Lidar1")(
    (extrinsics): Transform3D(
        (translation): Vector3D(1.1, 2.2, 3.3),
        (rotation): quaternion(1.1, 2.2, 3.3, 4.4)
```

(continues on next page)

(continued from previous page)

```
    )
)
```

dumps()

Dumps the sensor into a dict.

Returns A dict containing the information of the sensor.

Return type Dict[str, Any]

Examples

```
>>> # sensor is the object initialized from self.loads() method.
>>> sensor.dumps()
{
    'name': 'Lidar1',
    'type': 'LIDAR',
    'extrinsics': {'translation': {'x': 1.1, 'y': 2.2, 'z': 3.3},
    'rotation': {'w': 1.1, 'x': 2.2, 'y': 3.3, 'z': 4.4}
    }
}
```

set_extrinsics(*translation=(0, 0, 0)*, *rotation=(1, 0, 0, 0)*, ***, *matrix=None*)

Set the extrinsics of the sensor.

Parameters

- **translation** (*Iterable[float]*) – Translation parameters.
- **rotation** (*Union[Iterable[float], quaternion.quaternion]*) – Rotation in a sequence of [w, x, y, z] or numpy quaternion.
- **matrix** (*Optional[Union[Sequence[Sequence[float]], numpy.ndarray]]*) – A 3x4 or 4x4 transform matrix.

Return type None

Examples

```
>>> sensor.set_extrinsics(translation=translation, rotation=rotation)
>>> sensor
Lidar("Lidar1")(
    (extrinsics): Transform3D(
        (translation): Vector3D(1, 2, 3),
        (rotation): quaternion(1, 2, 3, 4)
    )
)
```

set_translation(*x*, *y*, *z*)

Set the translation of the sensor.

Parameters

- **x** (*float*) – The x coordinate of the translation.
- **y** (*float*) – The y coordinate of the translation.

- **z** (*float*) – The z coordinate of the translation.

Return type None

Examples

```
>>> sensor.set_translation(x=2, y=3, z=4)
>>> sensor
Lidar("Lidar1")(
    (extrinsics): Transform3D(
        (translation): Vector3D(2, 3, 4),
        ...
    )
)
```

set_rotation(*w=None*, *x=None*, *y=None*, *z=None*, *, *quaternion=None*)
Set the rotation of the sensor.

Parameters

- **w** (*Optional[float]*) – The w componet of the roation quaternion.
- **x** (*Optional[float]*) – The x componet of the roation quaternion.
- **y** (*Optional[float]*) – The y componet of the roation quaternion.
- **z** (*Optional[float]*) – The z componet of the roation quaternion.
- **quaternion** (*Optional[quaternion.quaternion]*) – Numpy quaternion representing the rotation.

Return type None

Examples

```
>>> sensor.set_rotation(2, 3, 4, 5)
>>> sensor
Lidar("Lidar1")(
    (extrinsics): Transform3D(
        ...
        (rotation): quaternion(2, 3, 4, 5)
    )
)
```

class tensorbay.sensor.sensor.**Lidar**(*name*)
Bases: *tensorbay.sensor.sensor.Sensor*

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.

Examples

```
>>> lidar = Lidar("Lidar1")
>>> lidar.set_extrinsics(translation=translation, rotation=rotation)
>>> lidar
Lidar("Lidar1")(
    (extrinsics): Transform3D(
        (translation): Vector3D(1, 2, 3),
        (rotation): quaternion(1, 2, 3, 4)
    )
)
```

Parameters **name** (*str*) –

Return type tensorbay.sensor.sensor._T

class tensorbay.sensor.sensor.**Radar**(*name*)

Bases: *tensorbay.sensor.sensor.Sensor*

Radar defines the concept of radar.

Radar is a detection system that uses radio waves to determine the range, angle, or velocity of objects.

Examples

```
>>> radar = Radar("Radar1")
>>> radar.set_extrinsics(translation=translation, rotation=rotation)
>>> radar
Radar("Radar1")(
    (extrinsics): Transform3D(
        (translation): Vector3D(1, 2, 3),
        (rotation): quaternion(1, 2, 3, 4)
    )
)
```

Parameters **name** (*str*) –

Return type tensorbay.sensor.sensor._T

class tensorbay.sensor.sensor.**Camera**(*name*)

Bases: *tensorbay.sensor.sensor.Sensor*

Camera defines the concept of camera.

Camera includes name, description, translation, rotation, cameraMatrix and distortionCoefficients.

Parameters **name** (*str*) –

Return type tensorbay.sensor.sensor._T

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*

Examples

```
>>> from tensorbay.geometry import Vector3D
>>> from numpy import quaternion
>>> camera = Camera('Camera1')
>>> translation = Vector3D(1, 2, 3)
>>> rotation = quaternion(1, 2, 3, 4)
>>> camera.set_extrinsics(translation=translation, rotation=rotation)
>>> camera.set_camera_matrix(fx=1.1, fy=1.1, cx=1.1, cy=1.1)
>>> camera.set_distortion_coefficients(p1=1.2, p2=1.2, k1=1.2, k2=1.2)
>>> camera
Camera("Camera1")(
    (extrinsics): Transform3D(
        (translation): Vector3D(1, 2, 3),
        (rotation): quaternion(1, 2, 3, 4)
    ),
    (intrinsics): CameraIntrinsics(
        (camera_matrix): CameraMatrix(
            (fx): 1.1,
            (fy): 1.1,
            (cx): 1.1,
            (cy): 1.1,
            (skew): 0
        ),
        (distortion_coefficients): DistortionCoefficients(
            (p1): 1.2,
            (p2): 1.2,
            (k1): 1.2,
            (k2): 1.2
        )
    )
)
```

classmethod loads(*contents*)

Loads a Camera from a dict containing the camera information.

Parameters **contents** (*Dict[str, Any]*) – A dict containing name, description, extrinsics and intrinsics.

Returns A *Camera* instance containing information from contents dict.

Return type tensorbay.sensor.sensor._T

Examples

```
>>> contents = {
...     "name": "Camera1",
...     "type": "CAMERA",
...     "extrinsics": {
...           "translation": {"x": 1, "y": 2, "z": 3},
...           "rotation": {"w": 1.0, "x": 2.0, "y": 3.0, "z": 4.0},
...     },
...     "intrinsics": {
...         "cameraMatrix": {"fx": 1, "fy": 1, "cx": 1, "cy": 1, "skew": 0},
...         "distortionCoefficients": {"p1": 1, "p2": 1, "k1": 1, "k2": 1},
...     },
... }
>>> Camera.loads(contents)
Camera("Camera1")(
        (extrinsics): Transform3D(
            (translation): Vector3D(1, 2, 3),
            (rotation): quaternion(1, 2, 3, 4)
        ),
        (intrinsics): CameraIntrinsics(
            (camera_matrix): CameraMatrix(
                (fx): 1,
                (fy): 1,
                (cx): 1,
                (cy): 1,
                (skew): 0
            ),
            (distortion_coefficients): DistortionCoefficients(
                (p1): 1,
                (p2): 1,
                (k1): 1,
                (k2): 1
            )
        )
    )
```

dumps()

Dumps the camera into a dict.

Returns A dict containing name, description, extrinsics and intrinsics.

Return type Dict[str, Any]

Examples

```
>>> camera.dumps()
{
    'name': 'Camera1',
    'type': 'CAMERA',
    'extrinsics': {
        'translation': {'x': 1, 'y': 2, 'z': 3},
        'rotation': {'w': 1.0, 'x': 2.0, 'y': 3.0, 'z': 4.0}
    },
    'intrinsics': {
        'cameraMatrix': {'fx': 1, 'fy': 1, 'cx': 1, 'cy': 1, 'skew': 0},
        'distortionCoefficients': {'p1': 1, 'p2': 1, 'k1': 1, 'k2': 1}
    }
}
```

set_camera_matrix(*fx=None*, *fy=None*, *cx=None*, *cy=None*, *skew=0*, ***, *matrix=None*)

Set camera matrix.

Parameters

- **fx** (*Optional[float]*) – The x axis focal length expressed in pixels.
- **fy** (*Optional[float]*) – The y axis focal length expressed in pixels.
- **cx** (*Optional[float]*) – The x coordinate of the so called principal point that should be in the center of the image.
- **cy** (*Optional[float]*) – The y coordinate of the so called principal point that should be in the center of the image.
- **skew** (*float*) – It causes shear distortion in the projected image.
- **matrix** (*Optional[Union[Sequence[Sequence[float]], numpy.ndarray]]*) – Camera matrix in 3x3 sequence.

Return type None

Examples

```
>>> camera.set_camera_matrix(fx=1.1, fy=2.2, cx=3.3, cy=4.4)
>>> camera
Camera("Camera1")(
    ...
    (intrinsics): CameraIntrinsics(
        (camera_matrix): CameraMatrix(
            (fx): 1.1,
            (fy): 2.2,
            (cx): 3.3,
            (cy): 4.4,
            (skew): 0
        ),
        ...
        )
    )
)
```

set_distortion_coefficients(***kwargs*)

Set distortion coefficients.

Parameters

- ****kwargs** – Float values to set distortion coefficients.
- **kwargs** (*float*) –

Raises **ValueError** – When intrinsics is not set yet.

Return type None

Examples

```
>>> camera.set_distortion_coefficients(p1=1.1, p2=2.2, k1=3.3, k2=4.4)
>>> camera
Camera("Camera1")(
    ...
    (intrinsics): CameraIntrinsics(
        ...
        (distortion_coefficients): DistortionCoefficients(
            (p1): 1.1,
            (p2): 2.2,
            (k1): 3.3,
            (k2): 4.4
        )
    )
)
```

class tensorbay.sensor.sensor.**FisheyeCamera**(*name*)

Bases: *tensorbay.sensor.sensor.Camera*

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.

Examples

```
>>> fisheye_camera = FisheyeCamera("FisheyeCamera1")
>>> fisheye_camera.set_extrinsics(translation=translation, rotation=rotation)
>>> fisheye_camera
FisheyeCamera("FisheyeCamera1")(
    (extrinsics): Transform3D(
        (translation): Vector3D(1, 2, 3),
        (rotation): quaternion(1, 2, 3, 4)
    )
)
```

Parameters **name** (*str*) –

Return type tensorbay.sensor.sensor._T

class tensorbay.sensor.sensor.**Sensors**

Bases: *tensorbay.utility.name.SortedNameList*[Union[*Radar*, *Lidar*, *FisheyeCamera*, *Camera*]]

This class represents all sensors in a *FusionSegment*.

classmethod loads(*contents*)

Loads a *Sensors* instance from the given contents.

Parameters contents (*List[Dict[str, Any]]*) – A list of dict containing the sensors information in a fusion segment, whose format should be like:

```
[
    {
        "name": <str>
        "type": <str>
        "extrinsics": {
            "translation": {
                "x": <float>
                "y": <float>
                "z": <float>
            },
            "rotation": {
                "w": <float>
                "x": <float>
                "y": <float>
                "z": <float>
            },
        },
        "intrinsics": {           --- only for cameras
            "cameraMatrix": {
                "fx": <float>
                "fy": <float>
                "cx": <float>
                "cy": <float>
                "skew": <float>
            }
            "distortionCoefficients": {
                "k1": <float>
                "k2": <float>
                "p1": <float>
                "p2": <float>
                ...
            }
        },
        "desctiption": <str>
    },
    ...
]
```

Returns The loaded *Sensors* instance.

Return type tensorbay.sensor.sensor._T

dumps()

Return the information of all the sensors.

Returns

A list of dict containing the information of all sensors:

```
[
    {
        "name": <str>
        "type": <str>
        "extrinsics": {
            "translation": {
                "x": <float>
                "y": <float>
                "z": <float>
            },
            "rotation": {
                "w": <float>
                "x": <float>
                "y": <float>
                "z": <float>
            },
        },
        "intrinsics": {           --- only for cameras
            "cameraMatrix": {
                "fx": <float>
                "fy": <float>
                "cx": <float>
                "cy": <float>
                "skew": <float>
            }
            "distortionCoefficients": {
                "k1": <float>
                "k2": <float>
                "p1": <float>
                "p2": <float>
                ...
            }
        },
        "desctiption": <str>
    },
    ...
]
```

Return type List[Dict[str, Any]]

1.24.6 tensorbay.utility

tensorbay.utility.attr

AttrsMixin and Field class.

`AttrsMixin` provides a list of special methods based on field configs.

`Field` is a class describing the attr related fields.

class `tensorbay.utility.attr.`**`Field`**(*, *is_dynamic*, *key*, *default*, *error_message*, *loader*, *dumper*)

Bases: `object`

A class to identify attr fields.

Parameters

- **is_dynamic** (*bool*) – Whether attr is a dynamic attr.
- **key** (*Union[str, None, Callable[[str], str]]*) – Display value of the attr in contents.
- **default** (*Any*) – Default value of the attr.
- **error_message** (*Optional[str]*) – The custom error message of the attr.
- **loader** (*Optional[Callable[[Any], Any]]*) – The custom loader of the attr.
- **dumper** (*Optional[Callable[[Any], Any]]*) – The custom dumper of the attr.

Return type None

`class tensorbay.utility.attr.`**BaseField**(*key*)

Bases: `object`

A class to identify fields of base class.

Parameters **key** (*Optional[str]*) – Display value of the attr.

Return type None

`class tensorbay.utility.attr.`**AttrsMixin**

Bases: `object`

AttrsMixin provides a list of special methods based on attr fields.

Examples

box2d: Box2DSubcatalog = attr(is_dynamic=True, key="BOX2D")

`tensorbay.utility.attr.`**attr**(*, *is_dynamic=False*, *key=<function <lambda>>*, *default=Ellipsis*, *error_message=None*, *loader=None*, *dumper=None*)

Return an instance to identify attr fields.

Parameters

- **is_dynamic** (*bool*) – Determine if this is a dynamic attr.
- **key** (*Union[str, None, Callable[[str], str]]*) – Display value of the attr in contents.
- **default** (*Any*) – Default value of the attr.
- **error_message** (*Optional[str]*) – The custom error message of the attr.
- **loader** (*Optional[Callable[[Any], Any]]*) – The custom loader of the attr.
- **dumper** (*Optional[Callable[[Any], Any]]*) – The custom dumper of the attr.

Raises *AttrError* – Dynamic attr cannot have default value.

Returns A *Field* instance containing all attr fields.

Return type Any

`tensorbay.utility.attr.`**attr_base**(*key=None*)

Return an instance to identify base class fields.

Parameters **key** (*Optional[str]*) – Display value of the attr.

Returns A `BaseField` instance containing all base class fields.

Return type Any

`tensorbay.utility.attr.upper(name)`
Convert the name value to uppercase.

Parameters **name** (*str*) – name of the attr.

Returns The uppercase value.

Return type str

`tensorbay.utility.attr.camel(name)`
Convert the name value to camelcase.

Parameters **name** (*str*) – name of the attr.

Returns The camelcase value.

Return type str

tensorbay.utility.common

Common_loads method, EqMixin class.

`common_loads()` is a common method for loading an object from a dict or a list of dict.

`EqMixin` is a mixin class to support __eq__() method, which compares all the instance variables.

`tensorbay.utility.common.common_loads(object_class, contents)`
A common method for loading an object from a dict or a list of dict.

Parameters

- **object_class** (*Type[tensorbay.utility.common._T]*) – The class of the object to be loaded.
- **contents** (*Any*) – The information of the object in a dict or a list of dict.

Returns The loaded object.

Return type tensorbay.utility.common._T

`class tensorbay.utility.common.EqMixin`
Bases: `object`

A mixin class to support __eq__() method.

The __eq__() method defined here compares all the instance variables.

`tensorbay.utility.common.locked(func)`
The decorator to add threading lock for methods.

Parameters **func** (*tensorbay.utility.common._CallableWithoutReturnValue*) – The method needs to add threading lock.

Returns The method with theading locked.

Return type tensorbay.utility.common._CallableWithoutReturnValue

tensorbay.utility.deprecated

Deprecated, KwargsDeprecated, DefaultValueDeprecated and Disable.

`Deprecated` is a decorator for deprecated functions.

`KwargsDeprecated` is a decorator for the function which has deprecated keyword arguments.

`DefaultValueDeprecated` is a decorator for the function which has deprecated argument default value.

`Disable` is a decorator for the function which is disabled temporarily.

class tensorbay.utility.deprecated.**Deprecated**(*, *since*, *removed_in=None*, *substitute=None*)

Bases: `object`

A decorator for deprecated functions.

Parameters

- **since** (*str*) – The version the function is deprecated.
- **removed_in** (*Optional[str]*) – The version the function will be removed in.
- **substitute** (*Union[None, str, Callable[[...], Any]]*) – The substitute function.

Return type None

class tensorbay.utility.deprecated.**KwargsDeprecated**(*keywords*, *, *since*, *removed_in=None*, *substitute=None*)

Bases: `object`

A decorator for the function which has deprecated keyword arguments.

Parameters

- **keywords** (*Tuple[str, ...]*) – The keyword arguments which need to be deprecated.
- **since** (*str*) – The version the keyword arguments are deprecated.
- **remove_in** – The version the keyword arguments will be removed in.
- **substitute** (*Optional[str]*) – The substitute usage.
- **removed_in** (*Optional[str]*) –

Return type None

class tensorbay.utility.deprecated.**DefaultValueDeprecated**(*keyword*, *, *since*, *removed_in=None*)

Bases: `object`

A decorator for the function which has deprecated argument default value.

Parameters

- **keyword** (*str*) – The argument keyword whose default value needs to be deprecated.
- **since** (*str*) – The version the keyword arguments are deprecated.
- **remove_in** – The version the keyword arguments will be removed in.
- **removed_in** (*Optional[str]*) –

Return type None

class tensorbay.utility.deprecated.**Disable**(*, *since*, *enabled_in*, *reason*)

Bases: `object`

A decorator for the function which is disabled temporarily.

Parameters

- **since** (*str*) – The version the function is disabled temporarily.
- **enabled_in** (*Optional[str]*) – The version the function will be enabled again.
- **reason** (*Optional[str]*) – The reason that the function is disabled temporarily.

Return type None

tensorbay.utility.file

mixin classes for local file and remote file.

class tensorbay.utility.file.**URL**(*url*, *updater*)

Bases: `object`

URL is a class used to get and update the url.

Parameters

- **url** (*str*) – The url.
- **updater** (*Callable[[], Optional[str]]*) – A function used to update the url.

Return type None

classmethod from_getter(*getter*, *updater*)

Create a URL instance from the given getter and updater.

Parameters

- **getter** (*Callable[[...], str]*) – The url getter of the file.
- **updater** (*Callable[[], Optional[str]]*) – The updater of the url.

Returns The URL instance which stores the url and the updater.

Return type *tensorbay.utility.file.URL*

update()

Update the url.

Return type None

get()

Get the url of the file.

Returns The url.

Return type str

class tensorbay.utility.file.**FileMixin**(*local_path*)

Bases: `tensorbay.utility.repr.ReprMixin`

FileMixin is a mixin class to mixin file related methods for local file.

Parameters **local_path** (*str*) – The file local path.

Return type None

path

The file local path.

get_checksum()

Get and cache the sha1 checksum of the local data.

Returns The sha1 checksum of the local data.

Return type str

get_url()
Return the url of the local data file.

Returns The url of the local data.

Return type str

open()
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.

Return type _io.BufferedReader

class tensorbay.utility.file.**RemoteFileMixin**(*remote_path*, ***, *url=None*, *cache_path=''*)
Bases: *tensorbay.utility.repr.ReprMixin*

RemoteFileMixin is a mixin class to mixin file related methods for remote file.

Parameters

- **local_path** – The file local path.
- **url** (*Optional[tensorbay.utility.file.URL]*) – The URL instance used to get and update url.
- **cache_path** (*str*) – The path to store the cache.
- **remote_path** (*str*) –

Return type None

path
The file local path.

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

Returns The url of the data.

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

Return type str

open()
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.

Return type Union[tensorbay.utility.requests.UserResponse, _io.BufferedReader]

tensorbay.utility.itertools

Utility tools about iteration.

`tensorbay.utility.itertools.`**`chunked`**(*iterable*, *n*)

Break an iterable instance into tuples of length n.

Parameters

- **iterable** (*Iterable[tensorbay.utility.itertools._T]*) – The input iterable instance which needs to be breaked into tuples of length n.
- **n** (*int*) – The length of each yielded tuples.

Yields The tuples of length n.

Return type Iterator[Tuple[tensorbay.utility.itertools._T, . . .]]

Examples

```
>>> list(chunked(range(9), 3))
[(0, 1, 2), (3, 4, 5), (6, 7, 8)]
```

The last yielded tuple may have fewer than n items if the length of the input iterable instance is not divisible by n:

```
>>> list(chunked(range(10), 3))
[(0, 1, 2), (3, 4, 5), (6, 7, 8), (9,)]
```

tensorbay.utility.name

NameMixin, SortedNameList and NameList.

NameMixin is a mixin class for instance which has immutable name and mutable description.

SortedNameList is a sorted sequence class which contains *NameMixin*. It is maintained in sorted order according to the 'name' of *NameMixin*.

NameList is a list of named elements, supports searching the element by its name.

class `tensorbay.utility.name.`**`NameMixin`**(*name*, *description=''*)

Bases: *tensorbay.utility.attr.AttrsMixin*, *tensorbay.utility.repr.ReprMixin*

A mixin class for instance which has immutable name and mutable description.

Parameters

- **name** (*str*) – Name of the class.
- **description** (*str*) – Description of the class.

Return type None

name

Name of the class.

class `tensorbay.utility.name.`**`NameList`**(*values=()*)

Bases: *tensorbay.utility.user.UserSequence*[tensorbay.utility.name._T]

NameList is a list of named elements, supports searching the element by its name.

keys()

Get all element names.

Returns A tuple containing all elements names.

Return type Tuple[str, …]

append(*value*)

Append element to the end of the NameList.

Parameters value (*tensorbay.utility.name._T*) – Element to be appended to the NameList.

Raises KeyError – When the name of the appending object already exists in the NameList.

Return type None

class tensorbay.utility.name.**SortedNameList**

Bases: *tensorbay.utility.user.UserSequence*[tensorbay.utility.name._T]

SortedNameList is a sorted sequence which contains element with name.

It is maintained in sorted order according to the 'name' attr of the element.

add(*value*)

Store element in name sorted list.

Parameters value (*tensorbay.utility.name._T*) – The element needs to be added to the list.

Raises KeyError – If the name of the added value exists in the list.

Return type None

keys()

Get all element names.

Returns A tuple containing all elements names.

Return type Tuple[str, …]

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.**ReprType**(*value*)

Bases: enum.Enum

ReprType is an enumeration type.

It defines the repr strategy type and includes 'INSTANCE', 'SEQUENCE' and 'MAPPING'.

class tensorbay.utility.repr.**ReprMixin**

Bases: object

ReprMixin provides customized repr config and method.

tensorbay.utility.type

TypeEnum, TypeMixin and TypeRegister.

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*.

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: Type[Any]

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: tensorbay.utility.type._T

Get the corresponding TypeEnum.

Returns The corresponding TypeEnum.

class tensorbay.utility.type.**TypeRegister**(*enum*)

Bases: Generic[tensorbay.utility.type._T]

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.**UserSequence**(**args*, ***kwds*)

Bases: Sequence[tensorbay.utility.user._T], *tensorbay.utility.repr.ReprMixin*

UserSequence is a user-defined wrapper around sequence objects.

index(*value*, *start=0*, *stop=9223372036854775807*)

Return the first index of the value.

Parameters

- **value** (*tensorbay.utility.user._T*) – The value to be found.
- **start** (*int*) – The start index of the subsequence.

- **stop** (*int*) – The end index of the subsequence.

Returns The First index of value.

Return type int

count(*value*)

Return the number of occurrences of value.

Parameters **value** (*tensorbay.utility.user._T*) – The value to be counted the number of occurrences.

Returns The number of occurrences of value.

Return type int

class tensorbay.utility.user.**UserMutableSequence**(**args*, ***kwds*)

Bases: `MutableSequence[tensorbay.utility.user._T]`, *tensorbay.utility.user.UserSequence*`[tensorbay.utility.user._T]`

UserMutableSequence is a user-defined wrapper around mutable sequence objects.

insert(*index*, *value*)

Insert object before index.

Parameters

- **index** (*int*) – Position of the mutable sequence.
- **value** (*tensorbay.utility.user._T*) – Element to be inserted into the mutable sequence.

Return type None

append(*value*)

Append object to the end of the mutable sequence.

Parameters **value** (*tensorbay.utility.user._T*) – Element to be appended to the mutable sequence.

Return type None

clear()

Remove all items from the mutable sequence.

Return type None

extend(*values*)

Extend mutable sequence by appending elements from the iterable.

Parameters **values** (*Iterable[tensorbay.utility.user._T]*) – Elements to be Extended into the mutable sequence.

Return type None

reverse()

Reverse the items of the mutable sequence in place.

Return type None

pop(*index=- 1*)

Return the item at index (default last) and remove it from the mutable sequence.

Parameters **index** (*int*) – Position of the mutable sequence.

Returns Element to be removed from the mutable sequence.

Return type tensorbay.utility.user._T

remove(*value*)

Remove the first occurrence of value.

Parameters **value** (*tensorbay.utility.user._T*) – Element to be removed from the mutable sequence.

Return type None

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: tensorbay.utility.user._K*) → Optional[tensorbay.utility.user._V]

get(*key: tensorbay.utility.user._K*, *default: Union[tensorbay.utility.user._V, tensorbay.utility.user._T] = None*) → Union[tensorbay.utility.user._V, tensorbay.utility.user._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()

Return a new view of the (key, value) pairs in dict.

Returns The (key, value) pairs in dict.

Return type AbstractSet[Tuple[tensorbay.utility.user._K, tensorbay.utility.user._V]]

keys()

Return a new view of the keys in dict.

Returns The keys in dict.

Return type AbstractSet[tensorbay.utility.user._K]

values()

Return a new view of the values in dict.

Returns The values in dict.

Return type ValuesView[tensorbay.utility.user._V]

class tensorbay.utility.user.**UserMutableMapping**(**args*, ***kwds*)

Bases: MutableMapping[tensorbay.utility.user._K, tensorbay.utility.user._V], *tensorbay.utility.user.UserMapping*[tensorbay.utility.user._K, tensorbay.utility.user._V]

UserMutableMapping is a user-defined wrapper around mutable mapping objects.

clear()

Remove all items from the mutable mapping object.

Return type None

pop(*key: tensorbay.utility.user._K*) → tensorbay.utility.user._V

pop(*key: tensorbay.utility.user._K*, *default: Union[tensorbay.utility.user._V, tensorbay.utility.user._T] = <object object>*) → Union[tensorbay.utility.user._V, tensorbay.utility.user._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()

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.

Return type Tuple[tensorbay.utility.user._K, tensorbay.utility.user._V]

setdefault(*key*, *default=None*)

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** (*tensorbay.utility.user._K*) – The key for dict, which can be any immutable type.
- **default** (*Optional[tensorbay.utility.user._V]*) – 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.

Return type tensorbay.utility.user._V

update(*__m: Mapping[tensorbay.utility.user._K, tensorbay.utility.user._V]*, ***kwargs: tensorbay.utility.user._V*) → None

update(*__m: Iterable[Tuple[tensorbay.utility.user._K, tensorbay.utility.user._V]]*, ***kwargs: tensorbay.utility.user._V*) → None

update(***kwargs: tensorbay.utility.user._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.

1.24.7 tensorbay.apps

tensorbay.apps.sextant

Interact with sextant app at graviti marketplace.

class tensorbay.apps.sextant.**Evaluation**(*evaluation_id*, *created_at*, *benchmark*)

Bases: `object`

This class defines *Evaluation*.

Parameters

- **evaluation_id** (*str*) – Evaluation ID.

- **created_at** (*int*) – Created time of the evaluation.
- **benchmark** (Benchmark) – The *Benchmark*.

Return type None

get_result()

Get the result of the evaluation.

Returns The result dict of the evaluation.

Return type Dict[str, Any]

class tensorbay.apps.sextant.Benchmark(*name*, *benchmark_id*, *sextant*, *, *dataset_id=None*, *commit_id=None*, *categories=None*, *iou_threshold=None*, *customized_metrics=None*)

Bases: object

This class defines *Benchmark*.

Parameters

- **name** (*str*) – Name of the Benchmark.
- **dataset_id** (*Optional[str]*) – ID of the dataset on which this benchmark based.
- **commit_id** (*Optional[str]*) – ID of the commit which is used as the evaluation benchmark.
- **benchmark_id** (*str*) – Benchmark ID.
- **sextant** (Sextant) – The SextantClient.
- **categories** (*Optional[List[str]]*) – The needed evaluation categories, if not given, all categories will be used.
- **iou_threshold** (*Optional[float]*) – The IoU threshold.
- **customized_metrics** (*Optional[str]*) – Https url of the github repository.

Return type None

create_evaluation(*dataset_id*, *commit_id*)

Create an evaluation task.

Parameters

- **dataset_id** (*str*) – Id of the needed evaluation dataset.
- **commit_id** (*str*) – Id of the needed commit.

Returns The created evaluation instance.

Return type *tensorbay.apps.sextant.Evaluation*

list_evaluations()

List all evaluations.

Returns A list of evaluations.

Return type *tensorbay.client.lazy.PagingList*[*tensorbay.apps.sextant.Evaluation*]

class tensorbay.apps.sextant.Sextant(*access_key*, *url=''*)

Bases: *tensorbay.client.requests.Client*

This class defines *Sextant*.

Parameters

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

Return type None

create_benchmark(*name*, *dataset_id*, *commit_id*, ***, *categories=None*, *iou_threshold=None*, *customized_metrics=None*)

Create a benchmark with the given parameters.

Parameters

- **name** (*str*) – Name of the Benchmark.
- **dataset_id** (*str*) – ID of the dataset on which this benchmark based.
- **commit_id** (*str*) – ID of the commit which used as the evaluation benchmark.
- **categories** (*Optional[List[str]]*) – The needed evaluation categories, if not given, all categories will be used.
- **iou_threshold** (*Optional[float]*) – The IoU threshold.
- **customized_metrics** (*Optional[str]*) – Https url of the github repository.

Return type *tensorbay.apps.sextant.Benchmark*

Raises:# flake8: noqa: F402 ValueError: When iou_threshold and customized_metrics both given or both not given.

list_benchmarks()

List all benchmarks.

Returns The list of Benchmark instances.

Return type *tensorbay.client.lazy.PagingList*[*tensorbay.apps.sextant.Benchmark*]

get_benchmark(*name*)

Get a benchmark instance by name.

Parameters **name** (*str*) – Name of the Benchmark.

Returns The Benchmark instance with the given name.

Raises ***ResourceNotExistError*** – When the required benchmark does not exist.

Return type *tensorbay.apps.sextant.Benchmark*

1.24.8 tensorbay.exception

TensorBay cutoms exceptions.

The class hierarchy for TensorBay custom exceptions is:

```
+-- TensorBayException
    +-- ClientError
        +-- StatusError
        +-- DatasetTypeError
        +-- FrameError
        +-- ResponseError
            +-- AccessDeniedError
            +-- ForbiddenError
```

(continues on next page)

(continued from previous page)

```
            +-- InvalidParamsError
            +-- NameConflictError
            +-- RequestParamsMissingError
            +-- ResourceNotExistError
            +-- InternalServerError
            +-- UnauthorizedError
   +-- UtilityError
       +-- AttrError
    +-- TBRNError
    +-- OpenDatasetError
        +-- NoFileError
        +-- FileStructureError
```

`OperationError` is removed in version v1.13.0. Use *StatusError* or `ValueError` instead.

exception tensorbay.exception.**TensorBayException**(*message=None*)

Bases: `Exception`

This is the base class for TensorBay custom exceptions.

Parameters **message** (*Optional[str]*) – The error message.

exception tensorbay.exception.**ClientError**(*message=None*)

Bases: *tensorbay.exception.TensorBayException*

This is the base class for custom exceptions in TensorBay client module.

Parameters **message** (*Optional[str]*) –

exception tensorbay.exception.**StatusError**(*message=None*, ***, *is_draft=None*)

Bases: *tensorbay.exception.ClientError*

This class defines the exception for illegal status.

Parameters

- **is_draft** (*Optional[bool]*) – Whether the status is draft.
- **message** (*Optional[str]*) – The error message.

Return type None

exception tensorbay.exception.**DatasetTypeError**(*message=None*, ***, *dataset_name=None*, *is_fusion=None*)

Bases: *tensorbay.exception.ClientError*

This class defines the exception for incorrect type of the requested dataset.

Parameters

- **dataset_name** (*Optional[str]*) – The name of the dataset whose requested type is wrong.
- **is_fusion** (*Optional[bool]*) – Whether the dataset is a fusion dataset.
- **message** (*Optional[str]*) –

Return type None

exception tensorbay.exception.**FrameError**(*message=None*)

Bases: *tensorbay.exception.ClientError*

This class defines the exception for incorrect frame id.

Parameters **message** (*Optional[str]*) –

exception tensorbay.exception.**ResponseError**(*message=None*, *, *response=None*)

Bases: *tensorbay.exception.ClientError*

This class defines the exception for post response error.

Parameters

- **response** (*Optional[requests.models.Response]*) – The response of the request.
- **message** (*Optional[str]*) –

Return type None

response

The response of the request.

exception tensorbay.exception.**AccessDeniedError**(*message=None*, *, *response=None*)

Bases: *tensorbay.exception.ResponseError*

This class defines the exception for access denied response error.

Parameters

- **message** (*Optional[str]*) –
- **response** (*Optional[requests.models.Response]*) –

Return type None

exception tensorbay.exception.**ForbiddenError**(*message=None*, *, *response=None*)

Bases: *tensorbay.exception.ResponseError*

This class defines the exception for illegal operations Tensorbay forbids.

Parameters

- **message** (*Optional[str]*) –
- **response** (*Optional[requests.models.Response]*) –

Return type None

exception tensorbay.exception.**InvalidParamsError**(*message=None*, *, *response=None*, *param_name=None*, *param_value=None*)

Bases: *tensorbay.exception.ResponseError*

This class defines the exception for invalid parameters response error.

Parameters

- **response** (*Optional[requests.models.Response]*) – The response of the request.
- **param_name** (*Optional[str]*) – The name of the invalid parameter.
- **param_value** (*Optional[str]*) – The value of the invalid parameter.
- **message** (*Optional[str]*) –

Return type None

response

The response of the request.

exception tensorbay.exception.**NameConflictError**(*message=None*, *, *response=None*, *resource=None*, *identification=None*)

Bases: *tensorbay.exception.ResponseError*

This class defines the exception for name conflict response error.

Parameters

- **response** (*Optional[requests.models.Response]*) – The response of the request.
- **resource** (*Optional[str]*) – The type of the conflict resource.
- **identification** (*Optional[Union[int, str]]*) – The identification of the conflict resource.
- **message** (*Optional[str]*) –

Return type None

response
The response of the request.

exception tensorbay.exception.**RequestParamsMissingError**(*message=None*, *, *response=None*)
Bases: *tensorbay.exception.ResponseError*

This class defines the exception for request parameters missing response error.

Parameters

- **message** (*Optional[str]*) –
- **response** (*Optional[requests.models.Response]*) –

Return type None

exception tensorbay.exception.**ResourceNotExistError**(*message=None*, *, *response=None*, *resource=None*, *identification=None*)
Bases: *tensorbay.exception.ResponseError*

This class defines the exception for resource not existing response error.

Parameters

- **response** (*Optional[requests.models.Response]*) – The response of the request.
- **resource** (*Optional[str]*) – The type of the conflict resource.
- **identification** (*Optional[Union[int, str]]*) – The identification of the conflict resource.
- **response** – The response of the request.
- **message** (*Optional[str]*) –

Return type None

exception tensorbay.exception.**InternalServerError**(*message=None*, *, *response=None*)
Bases: *tensorbay.exception.ResponseError*

This class defines the exception for internal server error.

Parameters

- **message** (*Optional[str]*) –
- **response** (*Optional[requests.models.Response]*) –

Return type None

exception tensorbay.exception.**UnauthorizedError**(*message=None*, *, *response=None*)
Bases: *tensorbay.exception.ResponseError*

This class defines the exception for unauthorized response error.

Parameters

- **message** (*Optional[str]*) –
- **response** (*Optional[requests.models.Response]*) –

Return type None

exception tensorbay.exception.**OpenDatasetError**(*message=None*)

Bases: *tensorbay.exception.TensorBayException*

This is the base class for custom exceptions in TensorBay opendataset module.

Parameters **message** (*Optional[str]*) –

exception tensorbay.exception.**NoFileError**(*message=None*, ***, *pattern=None*)

Bases: *tensorbay.exception.OpenDatasetError*

This class defines the exception for no matching file found in the opendataset directory.

Parameters

- **pattern** (*Optional[str]*) – Glob pattern.
- **message** (*Optional[str]*) –

Return type None

exception tensorbay.exception.**FileStructureError**(*message=None*)

Bases: *tensorbay.exception.OpenDatasetError*

This class defines the exception for incorrect file structure in opendataset directory.

Parameters **message** (*Optional[str]*) –

exception tensorbay.exception.**ModuleImportError**(*message=None*, ***, *module_name=None*, *package_name=None*)

Bases: *tensorbay.exception.OpenDatasetError*, ModuleNotFoundError

This class defines the exception for import error of optional module in opendataset module.

Parameters

- **module_name** (*Optional[str]*) – The name of the optional module.
- **package_name** (*Optional[str]*) – The package name of the optional module.
- **message** (*Optional[str]*) –

Return type None

exception tensorbay.exception.**TBRNError**(*message=None*)

Bases: *tensorbay.exception.TensorBayException*

This class defines the exception for invalid TBRN.

Parameters **message** (*Optional[str]*) –

exception tensorbay.exception.**UtilityError**(*message=None*)

Bases: *tensorbay.exception.TensorBayException*

This is the base class for custom exceptions in TensorBay utility module.

Parameters **message** (*Optional[str]*) –

exception tensorbay.exception.**AttrError**(*message=None*)

Bases: *tensorbay.exception.UtilityError*

This class defines the exception for dynamic attr have default value.

Parameters **message** (*Optional[str]*) –

1.24.9 tensorbay.opendataset

AADB	AADB dataset.
AnimalPose5	5 Categories Animal-Pose dataset.
AnimalPose7	7 Categories Animal-Pose dataset.
AnimalsWithAttributes2	Animals with attributes 2 dataset.
BDD100K	BDD100K dataset.
BDD100K_10K	BDD100K_10K dataset.
BDD100K_MOT2020	BDD100K_MOT2020 dataset.
BDD100K_MOTS2020	BDD100K_MOTS2020 dataset.
BSTLD	BSTLD dataset.
BioIDFace	BioID Face dataset.
CACD	Cross-Age Celebrity Dataset (CACD) dataset.
CADC	CADC dataset.
CarConnection	Car Connection Picture dataset.
CCPD	CCPD dataset.
CCPDGreen	CCPDGreen dataset.
CIHP	CIHP dataset.
COCO2017	COCO2017 dataset.
COVIDChestXRay	COVID-chestxray dataset.
COVID_CT	COVID-CT dataset.
CoinImage	Coin Image dataset.
CompCars	CompCars dataset.
DAVIS2017SemiSupervised	DAVIS2017SemiSupervised dataset.
DAVIS2017Unsupervised	DAVIS2017Unsupervised dataset.
DeepRoute	DeepRoute dataset.
DogsVsCats	Dogs vs Cats dataset.
DownsampledImagenet	Downsampled Imagenet dataset.
Elpv	elpv dataset.
FLIC	FLIC dataset.
FSDD	Free Spoken Digit dataset.
Flower17	17 Category Flower dataset.
Flower102	102 Category Flower dataset.
HalpeFullBody	Halpe Full-Body Human Keypoints and HOI-Det dataset.
HardHatWorkers	Hard Hat Workers dataset.
HeadPoseImage	Head Pose Image dataset.
ImageEmotionAbstract	Image Emotion-abstract dataset.
ImageEmotionArtphoto	Image Emotion-art Photo dataset.
JHU_CROWD	JHU-CROWD++ dataset.
KenyanFoodOrNonfood	Kenyan Food or Nonfood dataset.
KenyanFoodType	Kenyan Food Type dataset.
KylbergTexture	Kylberg Texture dataset.
LIP	LIP dataset.
LISATrafficLight	LISA Traffic Light dataset.
LISATrafficSign	LISA Traffic Sign dataset.
LeedsSportsPose	Leeds Sports Pose dataset.
NeolixOD	Neolix OD dataset.

continues on next page

Table 1.11 – continued from previous page

Newsgroups20	20 Newsgroups dataset.
NightOwls	NightOwls dataset.
nuImages	nuImages dataset.
nuScenes	nuScenes dataset.
OxfordIIITPet	OxfordIIITPet dataset.
PASCALContext	PASCALContext dataset.
RP2K	RP2K dataset.
SCUT_FBP5500	SCUT_FBP5500 dataset.
SegTrack	SegTrack dataset.
SegTrack2	SegTrack2 dataset.
SVHN	SVHN dataset.
THCHS30	THCHS-30 dataset.
THUCNews	THUCNews dataset.
TLR	TLR dataset.
UAVDT	UAVDT dataset.
VOC2012ActionClassification	VOC2012ActionClassification dataset.
VOC2012Detection	VOC2012Detection dataset.
VOC2012Segmentation	VOC2012Segmentation dataset.
WIDER_FACE	WIDER FACE dataset.

tensorbay.opendataset.AADB

tensorbay.opendataset.**AADB**(*path*)

AADB dataset.

The file structure looks like:

<path>

AADB_newtest/ 0.500_farm1_487_20167490236_ae920475e2_b.jpg …

datasetImages_warp256/ farm1_441_19470426814_baae1eb396_b.jpg …

imgListFiles_label/ imgList<segment_name>Regression_<attribute_name>.txt …

Parameters **path** (*str*) – the root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.AnimalPose5

tensorbay.opendataset.**AnimalPose5**(*path*)

5 Categories Animal-Pose dataset.

The file structure should be like:

```
<path>
    keypoint_image_part1/
        cat/
            2007_000549.jpg
            2007_000876.jpg
            ...
```

(continues on next page)

(continued from previous page)

```
        ...
    PASCAL2011_animal_annotation/
        cat/
            2007_000549_1.xml
            2007_000876_1.xml
            2007_000876_2.xml
            ...
        ...
    animalpose_image_part2/
        cat/
            ca1.jpeg
            ca2.jpeg
            ...
        ...
    animalpose_anno2/
        cat/
            ca1.xml
            ca2.xml
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.AnimalPose7

tensorbay.opendataset.**AnimalPose7**(*path*)

7 Categories Animal-Pose dataset.

The file structure should be like:

```
<path>
    bndbox_image/
        antelope/
            Img-77.jpg
            ...
        ...
    bndbox_anno/
        antelope.json
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns loaded `Dataset` object.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.AnimalsWithAttributes2

`tensorbay.opendataset.AnimalsWithAttributes2(path)`

Animals with attributes 2 dataset.

The file structure should be like:

```
<path>
    classes.txt
    predicates.txt
    predicate-matrix-binary.txt
    JPEGImages/
        <classname>/
            <imagename>.jpg
        ...
    ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.BDD100K

`tensorbay.opendataset.BDD100K(path)`

BDD100K dataset.

The file structure should be like:

```
<path>
    bdd100k_images_100k/
        images/
            100k/
                test
                train
                val
        labels/
            det_20/
                det_train.json
                det_val.json
            lane/
                polygons/
                    lane_train.json
                    lane_val.json
            drivable/
                polygons/
                    drivable_train.json
                    drivable_val.json
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.BDD100K_10K

`tensorbay.opendataset.BDD100K_10K(path)`

BDD100K_10K dataset.

The file structure should be like:

```
<path>
    bdd100k_images_10k/
        images/
            10k/
                test/
                    cabc30fc-e7726578.jpg
                    ...
                train/
                    0a0a0b1a-7c39d841.jpg
                    ...
                val/
                    b1c9c847-3bda4659.jpg
                    ...
        labels/
            pan_seg/
                polygons/
                    pan_seg_train.json
                    pan_seg_val.json
                bitmasks/
                    train/
                        0a0a0b1a-7c39d841.png
                        ...
                    val/
                        b1c9c847-3bda4659.png
                        ...
            sem_seg/
                masks/
                    train/
                        0a0a0b1a-7c39d841.png
                        ...
                    val/
                        b1c9c847-3bda4659.png
                        ...
            ins_seg/
                bitmasks/
                    train/
                        0a0a0b1a-7c39d841.png
                        ...
                    val/
                        b1c9c847-3bda4659.png
                        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.BDD100K_MOT2020

`tensorbay.opendataset.BDD100K_MOT2020(path)`

BDD100K_MOT2020 dataset.

The file structure should be like:

```
<path>
    bdd100k_box_track_20/
        images/
            train/
                00a0f008-3c67908e/
                    00a0f008-3c67908e-0000001.jpg
                    ...
                ...
            val/
                b1c9c847-3bda4659/
                    b1c9c847-3bda4659-0000001.jpg
                    ...
                ...
            test/
                cabc30fc-e7726578/
                    cabc30fc-e7726578-0000001.jpg
                    ...
                ...
        labels/
            train/
                00a0f008-3c67908e.json
                ...
            val/
                b1c9c847-3bda4659.json
                ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.BDD100K_MOTS2020

`tensorbay.opendataset.BDD100K_MOTS2020(path)`

BDD100K_MOTS2020 dataset.

The file structure should be like:

```
<path>
    bdd100k_seg_track_20/
        images/
            seg_track_20/
                test/
                    cabc30fc-e7726578/
                        cabc30fc-e7726578-0000001.jpg
                        ...
```

(continues on next page)

(continued from previous page)

```
            ...
        train/
            000d4f89-3bcbe37a/
                000d4f89-3bcbe37a-0000001.jpg
                ...
            ...
        val/
            b1c9c847-3bda4659/
                b1c9c847-3bda4659-0000001.jpg
                ...
            ...
    labels/
        seg_track_20/
            bitmasks/
                train/
                    000d4f89-3bcbe37a/
                        000d4f89-3bcbe37a-0000001.png
                        ...
                    ...
                val/
                    b1c9c847-3bda4659/
                        b1c9c847-3bda4659-0000001.png
                        ...
                    ...
            polygons/
                train/
                    000d4f89-3bcbe37a.json
                    ...
                val/
                    b1c9c847-3bda4659.json
                    ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.BSTLD

`tensorbay.opendataset.BSTLD(path)`

BSTLD dataset.

The file structure should be like:

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

(continues on next page)

(continued from previous page)

```
        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
```

Parameters **path** (*str*) – The root directory of the dataset.

Raises *ModuleImportError* – When the module "yaml" can not be found.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.BioIDFace

`tensorbay.opendataset.BioIDFace(path)`

BioID Face dataset.

The folder structure should be like:

```
<path>
    BioID-FaceDatabase-V1.2/
        BioID_0000.eye
        BioID_0000.pgm
        ...
    points_20/
        bioid_0000.pts
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.CACD

`tensorbay.opendataset.CACD(path)`

Cross-Age Celebrity Dataset (CACD) dataset.

The file structure should be like:

```
<path>
    CACD2000/
        14_Aaron_Johnson_0001.jpg
        ...
    celebrity2000.mat
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.CADC

`tensorbay.opendataset.CADC(path)`

CADC dataset.

The file structure should be like:

```
<path>
    2018_03_06/
        0001/
            3d_ann.json
            labeled/
                image_00/
                    data/
                        0000000000.png
                        0000000001.png
                        ...
                    timestamps.txt
                ...
                image_07/
                    data/
                    timestamps.txt
                lidar_points/
                    data/
                    timestamps.txt
                novatel/
                    data/
                    dataformat.txt
                    timestamps.txt
        ...
        0018/
        calib/
            00.yaml
            01.yaml
            02.yaml
            03.yaml
            04.yaml
            05.yaml
            06.yaml
            07.yaml
            extrinsics.yaml
            README.txt
    2018_03_07/
    2019_02_27/
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded *~tensorbay.dataset.dataset.FusionDataset* instance.

Return type *tensorbay.dataset.dataset.FusionDataset*

tensorbay.opendataset.CarConnection

tensorbay.opendataset.**CarConnection**(*path*)

Car Connection Picture dataset.

The file structure should be like:

```
<path>
    <imagename>.jpg
    ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.CCPD

tensorbay.opendataset.**CCPD**(*path*)

CCPD dataset.

The file structure should be like:

```
<path>
    ccpd_np/
        1005.jpg
        1019.jpg
        ...
    ccpd_base/
        00205459770115-90_85-352&516_448&547-                 444&547_368&549_364&
↪517_440&515-0_0_22_10_26_29_24-128-7.jpg
        00221264367816-91_91-283&519_381&553-                 375&551_280&552_285&
↪514_380&513-0_0_7_26_17_33_29-95-9.jpg
        ...
    ccpd_blur/
    ccpd_challenge/
    ccpd_db/
    ccpd_fn/
    ccpd_rotate/
    ccpd_tilt/
    ccpd_weather/
    LICENSE
    README.md
    splits/
        ccpd_blur.txt
        ccpd_challenge.txt
        ccpd_db.txt
        ccpd_fn.txt
        ccpd_rotate.txt
        ccpd_tilt.txt
        test.txt
        train.txt
        val.txt
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns class: *~tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.CCPDGreen

tensorbay.opendataset.**CCPDGreen**(*path*)

CCPDGreen dataset.

The file structure should be like:

```
<path>
    ccpd_green/
        train/
        test/
        val/
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns class: *~tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.CIHP

tensorbay.opendataset.**CIHP**(*path*)

CIHP dataset.

The file structure should be like:

```
<path>
    Testing/
        Images/
            0000002.jpg
            ...
        test_id.txt
    Training/
        Images/
            0000006.jpg
            ...
        Category_ids/
            0000006.png
            ...
        Instance_ids/
            0000006.png
            ...
        train_id.txt
    Validation/
        Images/
            0000001.jpg
            ...
        Category_ids/
            0000001.png
```

(continues on next page)

(continued from previous page)

```
        ...
    Instance_ids/
        0000001.png
        ...
    val_id.txt
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.COCO2017

tensorbay.opendataset.**COCO2017**(*path*)

COCO2017 dataset.

The file structure should be like:

```
<path>
    annotations/
        panoptic_train2017/
            000000116037.png
            000000116040.png
            ...
        panoptic_val2017/
        instances_train2017.json
        instances_val2017.json
        panoptic_train2017.json
        panoptic_val2017.json
        person_keypoints_train2017.json
        person_keypoints_val2017.json
    train2017/
        000000116026.jpg
        000000116031.jpg
        ...
    test2017/
    val2017/
    unlabeled2017/
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns class: *~tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.COVIDChestXRay

`tensorbay.opendataset.COVIDChestXRay(path)`

COVID-chestxray dataset.

The file structure should be like:

```
<path>
    images/
        0a7faa2a.jpg
        000001-2.png
        000001-3.jpg
        1B734A89-A1BF-49A8-A1D3-66FAFA4FAC5D.jpeg
        ...
    volumes/
        coronacases_org_001.nii.gz
        ....
    metadata.csv
    ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.COVID_CT

`tensorbay.opendataset.COVID_CT(path)`

COVID-CT dataset.

The file structure should be like:

```
<path>
    Data-split/
        COVID/
            testCT_COVID.txt
            trainCT_COVID.txt
            valCT_COVID.txt
        NonCOVID/
            testCT_NonCOVID.txt
            trainCT_NonCOVID.txt
            valCT_NonCOVID.txt
    Images-processed/
        CT_COVID/
            ...
            2020.01.24.919183-p27-132.png
            2020.01.24.919183-p27-133.png
            ...
            PIIS0140673620303603%8.png
            ...
        CT_NonCOVID/
            0.jpg
            1%0.jog
```

(continues on next page)

(continued from previous page)

```
        ...
        91%1.jpg
        102.png
        ...
        2341.png
```

Parameters `path` (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.CoinImage

`tensorbay.opendataset.`**`CoinImage`**(*path*)

Coin Image dataset.

The file structure should be like:

```
<path>
    classes.csv
    <imagename>.png
    ...
```

Parameters `path` (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.CompCars

`tensorbay.opendataset.`**`CompCars`**(*path*)

CompCars 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
```

(continues on next page)

(continued from previous page)

```
                    ...
                ...
            ...
        ...
    misc/
        attributes.txt
        car_type.mat
        make_model_name.mat
    train_test_split/
        classification/
            train.txt
            test.txt
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.DAVIS2017SemiSupervised

`tensorbay.opendataset.`**`DAVIS2017SemiSupervised`**(*path*)

DAVIS2017SemiSupervised dataset.

The file structure should be like:

```
<path>
    Annotations/
        480p/
            aerobatics/
                00000.png
            bear/
            ...
        Full-Resolution/
            aerobatics/
                00000.png
            bear/
            ...
    Annotations_semantics/
        480p/
            aerobatics/
                00000.png
            bear/
            ...
        Full-Resolution/
            aerobatics/
                00000.png
            bear/
            ...
    JPEGImages/
        480p/
            aerobatics/
```

(continues on next page)

(continued from previous page)

```
                00000.jpg
                00001.jpg
                ...
            bear/
            ...
        Full-Resolution/
            aerobatics/
                00000.jpg
                00001.jpg
                ...
            bear/
            ...
    ImageSets/
        2017/
            test-challenge.txt
            test-dev.txt
            train.txt
            val.txt
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns class: *~tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.DAVIS2017Unsupervised

tensorbay.opendataset.**DAVIS2017Unsupervised**(*path*)

DAVIS2017Unsupervised dataset.

The file structure should be like:

```
<path>
    Annotations_unsuperviseds/
        480p/
            bear/
                00000.png
                00001.png
                ...
            ...
        Full-Resolution/
            bear/
                00000.png
                00001.png
                ...
            ...
    JPEGImages/
        480p/
            aerobatics/
                00000.jpg
                00001.jpg
                ...
```

(continues on next page)

(continued from previous page)

```
            bear/
                00000.jpg
                00001.jpg
                ...
            ...
        Full-Resolution/
            aerobatics/
                00000.jpg
                00001.jpg
                ...
            bear/
                00000.jpg
                00001.jpg
                ...
            ...
    ImageSets/
        2017/
            train.txt
            val.txt
        2019/
            test-challenge.txt
            test-dev.txt
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns class: *~tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.DeepRoute

tensorbay.opendataset.**DeepRoute**(*path*)

DeepRoute dataset.

The file structure should be like:

```
<path>
    pointcloud/
        00001.bin
        00002.bin
        ...
        10000.bin
    groundtruth/
        00001.txt
        00002.txt
        ...
        10000.txt
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.DogsVsCats

tensorbay.opendataset.**DogsVsCats**(*path*)

Dogs vs Cats dataset.

The file structure should be like:

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

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.DownsampledImagenet

tensorbay.opendataset.**DownsampledImagenet**(*path*)

Downsampled Imagenet dataset.

The file structure should be like:

```
<path>
    valid_32x32/
        <imagename>.png
        ...
    valid_64x64/
        <imagename>.png
        ...
    train_32x32/
        <imagename>.png
        ...
    train_64x64/
        <imagename>.png
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.Elpv

`tensorbay.opendataset.Elpv(path)`

elpv dataset.

The file structure should be like:

```
<path>
    labels.csv
    images/
        cell0001.png
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.FLIC

`tensorbay.opendataset.FLIC(path)`

FLIC dataset.

The folder structure should be like:

```
<path>
    exampls.mat
    images/
        2-fast-2-furious-00003571.jpg
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Raises ***ModuleImportError*** – When the module "scipy" can not be found.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.FSDD

`tensorbay.opendataset.FSDD(path)`

Free Spoken Digit dataset.

The file structure should be like:

```
<path>
    recordings/
        0_george_0.wav
        0_george_1.wav
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.Flower17

`tensorbay.opendataset.`**`Flower17`**(*path*)

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.

The file structure should be like:

```
<path>
    jpg/
        image_0001.jpg
        ...
    datasplits.mat
```

Parameters **path** (*str*) – The root directory of the dataset.

Raises ***ModuleImportError*** – When the module "scipy" can not be found.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.Flower102

`tensorbay.opendataset.`**`Flower102`**(*path*)

102 Category Flower dataset.

The file structure should be like:

```
<path>
    jpg/
        image_00001.jpg
        ...
    imagelabels.mat
    setid.mat
```

Parameters **path** (*str*) – The root directory of the dataset.

Raises ***ModuleImportError*** – When the module "scipy" can not be found.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.HalpeFullBody

`tensorbay.opendataset.HalpeFullBody(path)`

Halpe Full-Body Human Keypoints and HOI-Det dataset.

The folder structure should be like:

```
<path>
    halpe_train_v1.json
    halpe_val_v1.json
    hico_20160224_det/
        images/
            train2015/
                HICO_train2015_00000001.jpg
                ...
    val2017/
        000000000139.jpg
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.HardHatWorkers

`tensorbay.opendataset.HardHatWorkers(path)`

Hard Hat Workers dataset.

The file structure should be like:

```
<path>
    annotations/
        hard_hat_workers0.xml
        ...
    images/
        hard_hat_workers0.png
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.HeadPoseImage

`tensorbay.opendataset.`**`HeadPoseImage`**(*path*)

Head Pose Image 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/
```

Parameters **`path`** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.ImageEmotionAbstract

`tensorbay.opendataset.`**`ImageEmotionAbstract`**(*path*)

Image Emotion-abstract dataset.

The file structure should be like:

```
<path>
    ABSTRACT_groundTruth.csv
    abstract_xxxx.jpg
    ...
```

Parameters **`path`** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.ImageEmotionArtphoto

`tensorbay.opendataset.`**`ImageEmotionArtphoto`**(*path*)

Image Emotion-art Photo dataset.

The file structure should be like:

```
<path>
    <filename>.jpg
    ...
```

Parameters **`path`** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.JHU_CROWD

`tensorbay.opendataset.JHU_CROWD(path)`

JHU-CROWD++ dataset.

The file structure should be like:

```
<path>
    train/
        images/
            0000.jpg
            ...
        gt/
            0000.txt
            ...
        image_labels.txt
    test/
    val/
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.KenyanFoodOrNonfood

`tensorbay.opendataset.KenyanFoodOrNonfood(path)`

Kenyan Food or Nonfood dataset.

The file structure should be like:

```
<path>
    images/
        food/
            236171947206673742.jpg
            ...
        nonfood/
            168223407.jpg
            ...
    data.csv
    split.py
    test.txt
    train.txt
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.KenyanFoodType

`tensorbay.opendataset.KenyanFoodType(path)`

Kenyan Food Type 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
            ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.KylbergTexture

`tensorbay.opendataset.KylbergTexture(path)`

Kylberg Texture dataset.

The file structure should be like:

```
<path>
    originalPNG/
        <imagename>.png
        ...
    withoutRotateAll/
        <imagename>.png
        ...
    RotateAll/
        <imagename>.png
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.LIP

`tensorbay.opendataset.LIP(path)`

LIP dataset.

The file structure should be like:

```
<path>
    Testing_images/
        testing_images/
            315_462476.jpg
            ...
        test_id.txt
    TrainVal_images/
        TrainVal_images/
            train_images/
                77_471474.jpg
                ...
            val_images/
                36_453991.jpg
                ...
        train_id.txt
        val_id.txt
    TrainVal_parsing_annotations/
        TrainVal_parsing_annotations/
            train_segmentations/
                77_471474.png
                ...
            val_segmentations/
                36_453991.png
                ...
    TrainVal_pose_annotations/
        lip_train_set.csv
        lip_val_set.csv
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded *~tensorbay.dataset.dataset.Dataset* instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.LISATrafficLight

`tensorbay.opendataset.LISATrafficLight(path)`

LISA Traffic Light 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/
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Raises `FileStructureError` – When frame number is discontinuous.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.LISATrafficSign

`tensorbay.opendataset.LISATrafficSign(path)`

LISA Traffic Sign dataset.

The file structure should be like:

```
<path>
    readme.txt
    allAnnotations.csv
    categories.txt
    datasetDescription.pdf
    videoSources.txt
    aiua120214-0/
        frameAnnotations-DataLog02142012_external_camera.avi_annotations/
            diff.txt
            frameAnnotations.bak
            frameAnnotations.bak2
            frameAnnotations.csv
            keepRight_1330547092.avi_image10.png
            keepRight_1330547092.avi_image11.png
            keepRight_1330547092.avi_image12.png
            ...
    aiua120214-1/
        frameAnnotations-DataLog02142012_001_external_camera.avi_annotations/
    aiua120214-2/
        frameAnnotations-DataLog02142012_002_external_camera.avi_annotations/
    aiua120306-0/
        frameAnnotations-DataLog02142012_002_external_camera.avi_annotations/
    aiua120306-1/
        frameAnnotations-DataLog02142012_003_external_camera.avi_annotations/
    vid0/
        frameAnnotations-vid_cmp2.avi_annotations/
    vid1/
        frameAnnotations-vid_cmp1.avi_annotations/
    vid10/
        frameAnnotations-MVI_0122.MOV_annotations/
    vid11/
        frameAnnotations-MVI_0123.MOV_annotations/
    vid2/
        frameAnnotations-vid_cmp2.avi_annotations/
    vid3/
        frameAnnotations-vid_cmp2.avi_annotations/
    vid4/
        frameAnnotations-vid_cmp2.avi_annotations/
    vid5/
        frameAnnotations-vid_cmp2.avi_annotations/
    vid6/
        frameAnnotations-MVI_0071.MOV_annotations/
    vid7/
        frameAnnotations-MVI_0119.MOV_annotations/
    vid8/
        frameAnnotations-MVI_0120.MOV_annotations/
    vid9/
```

(continues on next page)

(continued from previous page)

```
        frameAnnotations-MVI_0121.MOV_annotations/
    negatives/
        negativePics/
        negatives.dat
    tools/
        evaluateDetections.py
        extractAnnotations.py
        mergeAnnotationFiles.py
        splitAnnotationFiles.py
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.LeedsSportsPose

tensorbay.opendataset.**LeedsSportsPose**(*path*)

Leeds Sports Pose dataset.

The folder structure should be like:

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

Parameters **path** (*str*) – The root directory of the dataset.

Raises ***ModuleImportError*** – When the module "scipy" can not be found.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.NeolixOD

tensorbay.opendataset.**NeolixOD**(*path*)

Neolix OD dataset.

The file structure should be like:

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

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.Newsgroups20

`tensorbay.opendataset.`**`Newsgroups20`**(*path*)

20 Newsgroups dataset.

The folder structure should be like:

```
<path>
    20news-18828/
        alt.atheism/
            49960
            51060
            51119
            51120
            ...
        comp.graphics/
        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/
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.NightOwls

`tensorbay.opendataset.NightOwls(path)`

NightOwls 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
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.nuImages

`tensorbay.opendataset.nuImages(path)`

nuImages dataset.

The file structure should be like:

```
<path>
    nuimages-v1.0-all-metadata/
        v1.0-mini/
            attribute.json
            calibrated_sensor.json
            category.json
            ego_pose.json
            instance.json
            log.json
            object_ann.json
            sample_data.json
            sample.json
            sensor.json
            surface_ann.json
        v1.0-test/
            ...
        v1.0-train/
            ...
        v1.0-val/
            ...
    samples/
```

(continues on next page)

(continued from previous page)

```
        CAM_BACK/
        CAM_BACK_LEFT/
        CAM_BACK_RIGHT/
        CAM_FRONT/
        CAM_FRONT_LEFT/
        CAM_FRONT_RIGHT/
    sweeps/
        CAM_BACK/
        CAM_BACK_LEFT/
        CAM_BACK_RIGHT/
        CAM_FRONT/
        CAM_FRONT_LEFT/
        CAM_FRONT_RIGHT/
    nuimages-v1.0-mini/
        samples/
            ...
        sweeps/
            ...
        v1.0-mini/
           ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.FusionDataset*

tensorbay.opendataset.nuScenes

tensorbay.opendataset.**nuScenes**(*path*)

nuScenes dataset.

The file structure should be like:

```
<path>
    v1.0-mini/
        maps/
            36092f0b03a857c6a3403e25b4b7aab3.png
            ...
        samples/
            CAM_BACK/
            CAM_BACK_LEFT/
            CAM_BACK_RIGHT/
            CAM_FRONT/
            CAM_FRONT_LEFT/
            CAM_FRONT_RIGHT/
            LIDAR_TOP/
            RADAR_BACK_LEFT/
            RADAR_BACK_RIGHT/
            RADAR_FRONT/
            RADAR_FRONT_LEFT/
            RADAR_FRONT_RIGHT/
```

(continues on next page)

(continued from previous page)

```
        sweeps/
            CAM_BACK/
            CAM_BACK_LEFT/
            CAM_BACK_RIGHT/
            CAM_FRONT/
            CAM_FRONT_LEFT/
            CAM_FRONT_RIGHT/
            LIDAR_TOP/
            RADAR_BACK_LEFT/
            RADAR_BACK_RIGHT/
            RADAR_FRONT/
            RADAR_FRONT_LEFT/
            RADAR_FRONT_RIGHT/
        v1.0-mini/
            attribute.json
            calibrated_sensor.json
            category.json
            ego_pose.json
            instance.json
            log.json
            map.json
            sample_annotation.json
            sample_data.json
            sample.json
            scene.json
            sensor.json
            visibility.json
    v1.0-test/
        maps/
        samples/
        sweeps/
        v1.0-test/
    v1.0-trainval/
        maps/
        samples/
        sweeps/
        v1.0-trainval/
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `FusionDataset` instance.

Return type *tensorbay.dataset.dataset.FusionDataset*

tensorbay.opendataset.OxfordIIITPet

`tensorbay.opendataset.OxfordIIITPet(path)`

OxfordIIITPet dataset.

The file structure should be like:

```
<path>
    annotations/
        trimaps/
            Bombay_113.png
            Bombay_114.png
            ...
        xmls/
            Birman_174.xml
            Birman_175.xml
            ...
        list.txt
        test.txt
        trainval.txt
        README
    images/
        Bombay_117.jpg
        Bombay_118.jpg
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns class: *~tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.PASCALContext

`tensorbay.opendataset.PASCALContext(mask_path, image_path)`

PASCALContext dataset.

The file structure should be like:

```
<mask_path>
    <image_name>.png
    ...

<image_path>
    <image_name>.jpg
    ...
```

Parameters

- **mask_path** (*str*) – The root directory of the dataset mask.
- **image_path** (*str*) – The root directory of the dataset image.

Returns class: *~tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.RP2K

tensorbay.opendataset.**RP2K**(*path*)

RP2K dataset.

The file structure of RP2K looks like:

```
<path>
    all/
        test/
            <catagory>/
                <image_name>.jpg
                ...
            ...
        train/
            <catagory>/
                <image_name>.jpg
                ...
            ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.SCUT_FBP5500

tensorbay.opendataset.**SCUT_FBP5500**(*path*)

SCUT_FBP5500 dataset.

The folder structure should be like:

```
<path>
    facial landmark/
        <file_name>.pts
        ...
    Images/
        <file_name>.jpg
        ...
    train_test_files/
        split_of_60%training and 40%testing/
            test.txt
            train.txt
        ...
    ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.SegTrack

`tensorbay.opendataset.SegTrack(path)`

SegTrack dataset.

The file structure of SegTrack looks like:

```
<path>
    birdfall2/
        birdfall2_00018.png
        ...
        ground-truth/
            birdfall2_00018.png
            ...
    cheetah/
        chasedeer_frame_0001.bmp
        ...
        ground-truth/
            chasedeer_00000.png
            ...
    girl/
        5117-8_70161.bmp
        ...
        ground-truth/
            0.bmp
            ...
    monkeydog/
        195.bmp
        ...
        ground-truth/
            Comp_00195.png
            ...
    parachute/
        parachute_00000.png
        ...
        ground-truth/
            parachute_00000.png
            ...
    penguin/
        penguin_00000.bmp
        ...
        ground_truth/
            penguin_00000.png
            ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.SegTrack2

`tensorbay.opendataset.SegTrack2(path)`

SegTrack2 dataset.

The file structure of SegTrack looks like:

```
<path>
    GroundTruth/
        bird_of_paradise/
            bird_of_paradise_00000.png
            ...
        bmx/
            1/
                bmx_06668.png
                ...
            2/
                bmx_06668.png
                ...
        ...
    JPEGImages/
        bird_of_paradise/
            bird_of_paradise_00000.png
            ...
        ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.SVHN

`tensorbay.opendataset.SVHN(path)`

SVHN dataset.

The file structure should be like:

```
<path>
    Cropped/
        extra_32x32.mat
        test_32x32.mat
        train_32x32.mat
    FullNumbers/
        extra/
            116507.png
            116508.png
            ...
            digitStruct.mat
            see_bboxes.m
        test/
        train/
```

Parameters **path** (*str*) – The root directory of the dataset.

Raises `ModuleImportError` – When the module "h5py" can not be found.

Returns class: ~*tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.THCHS30

`tensorbay.opendataset.`**`THCHS30`**(*path*)

THCHS-30 dataset.

The file structure should be like:

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

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.THUCNews

`tensorbay.opendataset.`**`THUCNews`**(*path*)

THUCNews dataset.

The folder structure should be like:

```
<path>
    <category>/
        0.txt
        1.txt
        2.txt
        3.txt
        ...
    <category>/
    ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.TLR

`tensorbay.opendataset.TLR(path)`

TLR 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
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.UAVDT

`tensorbay.opendataset.UAVDT(path)`

UAVDT dataset.

The "score", "in-view", "occlusion" fields in MOT Groundtruth file(`*_gt.txt`) are constant, and other fields in that file are the same with such fields in DET Groundtruth file (`*_gt_whole.txt`). Therefore, they are not included in the dataloader.

The Ignore Areas file(`*_gt_ignore.txt`) is useless, so they are not included in the dataloader neither.

The file structure of UAVDT looks like:

```
<path>
    M_attr/
        test/
            M0203_attr.txt
            ...
        train/
            M0101_attr.txt
            ...
    UAVDT_Benchmark_M/
        M0101/
            img000001.jpg
            ...
        ...
    UAV-benchmark-MOTD_v1.0/
        GT/
            M0101_gt_ignore.txt
            M0101_gt.txt
            M0101_gt_whole.txt
            ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

tensorbay.opendataset.VOC2012ActionClassification

tensorbay.opendataset.**VOC2012ActionClassification**(*path*)

VOC2012ActionClassification dataset.

The file structure should be like:

```
<path>
    Annotations/
        <image_name>.xml
        ...
    JPEGImages/
        <image_name>.jpg
        ...
    ImageSets/
        Action/
            train.txt
            val.txt
            ...
        ...
    ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns class: *~tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.VOC2012Detection

tensorbay.opendataset.**VOC2012Detection**(*path*)

VOC2012Detection dataset.

The file structure should be like:

```
<path>
    Annotations/
        <image_name>.xml
        ...
    JPEGImages/
        <image_name>.jpg
        ...
    ImageSets/
        Main/
            train.txt
            val.txt
            ...
        ...
    ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns class: ~*tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.VOC2012Segmentation

`tensorbay.opendataset.VOC2012Segmentation(path)`

VOC2012Segmentation dataset.

The file structure should be like:

```
<path>/
    JPEGImages/
        <image_name>.jpg
        ...
    SegmentationClass/
        <mask_name>.png
        ...
    SegmentationObject/
        <mask_name>.png
        ...
    ImageSets/
        Segmentation/
            train.txt
            val.txt
            ...
        ...
    ...
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns class: ~*tensorbay.dataset.dataset.Dataset* instance.

Return type Loaded

tensorbay.opendataset.WIDER_FACE

`tensorbay.opendataset.WIDER_FACE(path)`

WIDER FACE 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
                0_Parade_marchingband_1_1030.jpg
                ...
            1--Handshaking/
            ...
            59--people--driving--car/
            61--Street_Battle/
```

(continues on next page)

(continued from previous page)

```
    WIDER_val/
        ...
    WIDER_test/
        ...
    wider_face_split/
        wider_face_train_bbx_gt.txt
        wider_face_val_bbx_gt.txt
```

Parameters **path** (*str*) – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Return type *tensorbay.dataset.dataset.Dataset*

PYTHON MODULE INDEX

t

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

U

V

W

X

Y

Z