
TensorBay

Graviti

Aug 26, 2021

QUICK START

1 What can TensorBay SDK do?	3
Python Module Index	285
Index	287

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.

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
<i>Dogs vs Cats</i>	Topic: Dataset Management Data Type: Image Label Type: <i>Classification</i>
<i>20 Newsgroups</i>	Topic: Dataset Management Data Type: Text Label Type: <i>Classification</i>
<i>BSTLD</i>	Topic: Dataset Management Data Type: Image Label Type: <i>Box2D</i>
<i>Neolix OD</i>	Topic: Dataset Management Data Type: Point Cloud Label Type: <i>Box3D</i>
<i>Leeds Sports Pose</i>	Topic: Dataset Management Data Type: Image Label Type: <i>Keypoints2D</i>
<i>THCHS-30</i>	Topic: Dataset Management Data Type: Audio Label Type: <i>Sentence</i>
<i>Update Dataset</i>	Topic: Update Dataset
<i>Move And Copy</i>	Topic: Move And Copy

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

It takes the following steps to organize the “Dogs vs Cats” 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.

```
1 {
2   "CLASSIFICATION": {
3     "categories": [{ "name": "cat" }, { "name": "dog" }]
4   }
5 }
```

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

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.

```
1 #!/usr/bin/env python3
2 #
3 # Copyright 2021 Graviti. Licensed under MIT License.
4 #
5 # pylint: disable=invalid-name
6 # pylint: disable=missing-module-docstring
7
8 import os
```

(continues on next page)

```

9
10 from ...dataset import Data, Dataset
11 from ...label import Classification
12 from .._utility import glob
13
14 DATASET_NAME = "DogsVsCats"
15 _SEGMENTS = {"train": True, "test": False}
16
17
18 def DogsVsCats(path: str) -> Dataset:
19     """Dataloader of the `Dogs vs Cats` dataset.
20
21     .. _Dogs vs Cats: https://www.kaggle.com/c/dogs-vs-cats
22
23     The file structure should be like::
24
25         <path>
26             train/
27                 cat.0.jpg
28                 ...
29                 dog.0.jpg
30                 ...
31             test/
32                 1000.jpg
33                 1001.jpg
34                 ...
35
36     Arguments:
37         path: The root directory of the dataset.
38
39     Returns:
40         Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.
41
42     """
43     root_path = os.path.abspath(os.path.expanduser(path))
44     dataset = Dataset(DATASET_NAME)
45     dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))
46
47     for segment_name, is_labeled in _SEGMENTS.items():
48         segment = dataset.create_segment(segment_name)
49         image_paths = glob(os.path.join(root_path, segment_name, "*.jpg"))
50         for image_path in image_paths:
51             data = Data(image_path)
52             if is_labeled:
53                 data.label.classification = Classification(os.path.basename(image_
54 ↪ path)[:3])
55                 segment.append(data)
56
57     return dataset

```

See *Classification annotation* for more details.

Note: Since the *Dogs vs Cats dataloader* above is already included in TensorBay, so it uses relative import. However, the regular import should be used when writing a new dataloader.

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

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 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 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

It takes the following steps to organize the “BSTLD” 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.

```

1 {
2   "BOX2D": {
3     "categories": [
4       { "name": "Red" },
5       { "name": "RedLeft" },
6       { "name": "RedRight" },
7       { "name": "RedStraight" },
8       { "name": "RedStraightLeft" },
9       { "name": "Green" },
10      { "name": "GreenLeft" },
11      { "name": "GreenRight" },
12      { "name": "GreenStraight" },
13      { "name": "GreenStraightLeft" },
14      { "name": "GreenStraightRight" },
15      { "name": "Yellow" },
16      { "name": "off" }
17    ],
18    "attributes": [
19      {
20        "name": "occluded",
21        "type": "boolean"
22      }
23    ]
24  }
25 }
```

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

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.

```

1  #!/usr/bin/env python3
2  #
3  # Copyright 2021 Graviti. Licensed under MIT License.
4  #
5  # pylint: disable=invalid-name
6  # pylint: disable=missing-module-docstring
7
8  import os
9
10 from ...dataset import Data, Dataset
11 from ...exception import ModuleImportError
12 from ...label import LabeledBox2D
13
14 DATASET_NAME = "BSTLD"
15
16 _LABEL_FILENAME_DICT = {
17     "test": "test.yaml",
18     "train": "train.yaml",
19     "additional": "additional_train.yaml",
20 }
21
22
23 def BSTLD(path: str) -> Dataset:
24     """Dataloader of the `BSTLD` dataset.
25
26     .. _BSTLD: https://hci.iwr.uni-heidelberg.de/content/bosch-small-traffic-lights-
27     ↪ dataset
28
29     The file structure should be like::
30
31         <path>
32         rgb/
33             additional/
34                 2015-10-05-10-52-01_bag/
35                     <image_name>.jpg
36                     ...
37             ...
38             test/
39                 <image_name>.jpg
40                 ...
41             train/
42                 2015-05-29-15-29-39_arastradero_traffic_light_loop_bag/
43                     <image_name>.jpg
44                     ...
45             ...
46         test.yaml
47         train.yaml
48         additional_train.yaml

```

(continues on next page)

(continued from previous page)

```

49 Arguments:
50     path: The root directory of the dataset.
51
52 Raises:
53     ModuleNotFoundError: When the module "yaml" can not be found.
54
55 Returns:
56     Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.
57
58 """
59 try:
60     import yaml # pylint: disable=import-outside-toplevel
61 except ModuleNotFoundError as error:
62     raise ModuleNotFoundError(error.name, "pyyaml") from error # type: ignore[arg-
↪ type]
63
64 root_path = os.path.abspath(os.path.expanduser(path))
65
66 dataset = Dataset(DATASET_NAME)
67 dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))
68
69 for mode, label_file_name in _LABEL_FILENAME_DICT.items():
70     segment = dataset.create_segment(mode)
71     label_file_path = os.path.join(root_path, label_file_name)
72
73     with open(label_file_path, encoding="utf-8") as fp:
74         labels = yaml.load(fp, yaml.FullLoader)
75
76     for label in labels:
77         if mode == "test":
78             # the path in test label file looks like:
79             # /absolute/path/to/<image_name>.png
80             file_path = os.path.join(root_path, "rgb", "test", label["path"].rsplit(
↪ "/", 1)[-1])
81         else:
82             # the path in label file looks like:
83             # ./rgb/additional/2015-10-05-10-52-01_bag/<image_name>.png
84             file_path = os.path.join(root_path, *label["path"][2:].split("/"))
85         data = Data(file_path)
86         data.label.box2d = [
87             LabeledBox2D(
88                 box["x_min"],
89                 box["y_min"],
90                 box["x_max"],
91                 box["y_max"],
92                 category=box["label"],
93                 attributes={"occluded": box["occluded"]},
94             )
95             for box in label["boxes"]
96         ]
97         segment.append(data)
98

```

(continues on next page)

(continued from previous page)

99

```
return dataset
```

See [Box2D annotation](#) for more details.

Note: Since the [BSTLD dataloader](#) above is already included in TensorBay, so it uses relative import. However, the regular import should be used when writing a new dataloader.

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

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)
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 “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")
```

1.2.3 Leeds Sports Pose

This topic describes how to manage the *Leeds Sports Pose Dataset*, which is a dataset with *Keypoints2D* label(Fig. 1.2).

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)
```



Fig. 1.2: The preview of an image with labels from “Leeds Sports Pose”.

Create Dataset

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

Organize Dataset

It takes the following steps to organize the “Leeds Sports Pose” 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.

```

1 {
2   "KEYPOINTS2D": {
3     "keypoints": [
4       {
5         "number": 14,
6         "names": [
7           "Right ankle",
8           "Right knee",
9           "Right hip",
10          "Left hip",
11          "Left knee",
12          "Left ankle",
13          "Right wrist",
14          "Right elbow",
15          "Right shoulder",
16          "Left shoulder",
17          "Left elbow",
18          "Left wrist",
19          "Neck",
20          "Head top"
21        ],
22        "skeleton": [
23          [0, 1],
24          [1, 2],
25          [3, 4],
26          [4, 5],
27          [6, 7],
28          [7, 8],
29          [9, 10],
30          [10, 11],
31          [12, 13],
32          [12, 2],
33          [12, 3]
34        ],
35        "visible": "BINARY"
36      }
37    ]
38  }
39 }
```

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

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.

```
1  #!/usr/bin/env python3
2  #
3  # Copyright 2021 Graviti. Licensed under MIT License.
4  #
5  # pylint: disable=invalid-name
6  # pylint: disable=missing-module-docstring
7
8  import os
9
10 from ...dataset import Data, Dataset
11 from ...exception import ModuleImportError
12 from ...geometry import Keypoint2D
13 from ...label import LabeledKeypoints2D
14 from .._utility import glob
15
16 DATASET_NAME = "LeedsSportsPose"
17
18
19 def LeedsSportsPose(path: str) -> Dataset:
20     """Dataloader of the `Leeds Sports Pose`_ dataset.
21
22     .. _Leeds Sports Pose: https://sam.johnson.io/research/lsp.html
23
24     The folder structure should be like::
25
26         <path>
27             joints.mat
28             images/
29                 im0001.jpg
30                 im0002.jpg
31                 ...
32
33     Arguments:
34         path: The root directory of the dataset.
35
36     Raises:
37         ModuleImportError: When the module "scipy" can not be found.
38
39     Returns:
40         Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.
41
42     """
43     try:
```

(continues on next page)

(continued from previous page)

```

44     from scipy.io import loadmat # pylint: disable=import-outside-toplevel
45 except ModuleNotFoundError as error:
46     raise ModuleImportError(error.name) from error # type: ignore[arg-type]
47
48 root_path = os.path.abspath(os.path.expanduser(path))
49
50 dataset = Dataset(DATASET_NAME)
51 dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))
52 segment = dataset.create_segment()
53
54 mat = loadmat(os.path.join(root_path, "joints.mat"))
55
56 joints = mat["joints"].T
57 image_paths = glob(os.path.join(root_path, "images", "*.jpg"))
58 for image_path in image_paths:
59     data = Data(image_path)
60     data.label.keypoints2d = []
61     index = int(os.path.basename(image_path)[2:6]) - 1 # get image index from
    ↪ "im0001.jpg"
62
63     keypoints = LabeledKeypoints2D()
64     for keypoint in joints[index]:
65         keypoints.append( # pylint: disable=no-member # pylint issue #3131
66             Keypoint2D(keypoint[0], keypoint[1], int(not keypoint[2]))
67         )
68
69     data.label.keypoints2d.append(keypoints)
70     segment.append(data)
71 return dataset

```

See *Keypoints2D* annotation for more details.

Note: Since the *Leeds Sports Pose dataloader* above is already included in TensorBay, so it uses relative import. However, the regular import should be used when writing a new dataloader.

```

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

```

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")
```

1.2.4 Neolix OD

This topic describes how to manage the [Neolix OD](#) dataset, which is a dataset with *Box3D* label type (Fig. 1.3).

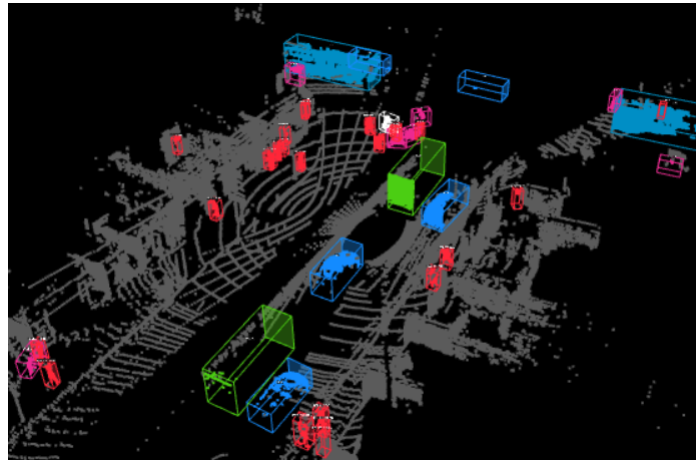


Fig. 1.3: The preview of a point cloud from “Neolix OD” with Box3D labels.

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

It takes the following steps to organize “Neolix OD” 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.

```

1 {
2   "BOX3D": {
3     "categories": [
4       { "name": "Adult" },
5       { "name": "Animal" },
6       { "name": "Barrier" },
7       { "name": "Bicycle" },
8       { "name": "Bicycles" },
9       { "name": "Bus" },
10      { "name": "Car" },
11      { "name": "Child" },
12      { "name": "Cyclist" },
13      { "name": "Motorcycle" },
14      { "name": "Motorcyclist" },
15      { "name": "Trailer" },
16      { "name": "Tricycle" },
17      { "name": "Truck" },
18      { "name": "Unknown" }
19    ],
20    "attributes": [
21      {
22        "name": "Alpha",
23        "type": "number",
24        "description": "Angle of view"
25      },
26      {
27        "name": "Occlusion",
28        "enum": [0, 1, 2],
29        "description": "It indicates the degree of occlusion of objects by other
↳ obstacles"
30      },
31      {
32        "name": "Truncation",
33        "type": "boolean",
34        "description": "It indicates whether the object is truncated by the edge
↳ of the image"
35      }
36    ]
37  }
38 }
```

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

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.

```

1  #!/usr/bin/env python3
2  #
3  # Copyright 2021 Graviti. Licensed under MIT License.
4  #
5  # pylint: disable=invalid-name
6  # pylint: disable=missing-module-docstring
7
8  import os
9
10 from quaternion import from_rotation_vector
11
12 from ...dataset import Data, Dataset
13 from ...label import LabeledBox3D
14 from ..utility import glob
15
16 DATASET_NAME = "NeolixOD"
17
18
19 def NeolixOD(path: str) -> Dataset:
20     """Dataloader of the `Neolix OD` dataset.
21
22     .. _Neolix OD: https://www.graviti.cn/dataset-detail/NeolixOD
23
24     The file structure should be like::
25
26         <path>
27             bins/
28                 <id>.bin
29             labels/
30                 <id>.txt
31             ...
32
33     Arguments:
34         path: The root directory of the dataset.
35
36     Returns:
37         Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.
38
39     """
40     root_path = os.path.abspath(os.path.expanduser(path))
41
42     dataset = Dataset(DATASET_NAME)
43     dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))
44     segment = dataset.create_segment()
45
46     point_cloud_paths = glob(os.path.join(root_path, "bins", "*.bin"))
47
48     for point_cloud_path in point_cloud_paths:
49         data = Data(point_cloud_path)

```

(continues on next page)

(continued from previous page)

```

50     data.label.box3d = []
51
52     point_cloud_id = os.path.basename(point_cloud_path)[:6]
53     label_path = os.path.join(root_path, "labels", f"{point_cloud_id}.txt")
54
55     with open(label_path, encoding="utf-8") as fp:
56         for label_value_raw in fp:
57             label_value = label_value_raw.rstrip().split()
58             label = LabeledBox3D(
59                 size=[float(label_value[10]), float(label_value[9]), float(label_
↪ value[8])],
60                 translation=[
61                     float(label_value[11]),
62                     float(label_value[12]),
63                     float(label_value[13]) + 0.5 * float(label_value[8]),
64                 ],
65                 rotation=from_rotation_vector((0, 0, float(label_value[14]))),
66                 category=label_value[0],
67                 attributes={
68                     "Occlusion": int(label_value[1]),
69                     "Truncation": bool(int(label_value[2])),
70                     "Alpha": float(label_value[3]),
71                 },
72             )
73             data.label.box3d.append(label)
74
75     segment.append(data)
76     return dataset

```

See *Box3D annotation* for more details.

Note: Since the *Neolix OD dataloader* above is already included in TensorBay, so it uses relative import. However, the regular import should be used when writing a new dataloader.

```

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

```

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 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.

```
1  #!/usr/bin/env python3
2  #
3  # Copyright 2021 Graviti. Licensed under MIT License.
4  #
5  # pylint: disable=invalid-name
6  # pylint: disable=missing-module-docstring
7
8  import os
9  from itertools import islice
10 from typing import List
11
12 from ...dataset import Data, Dataset
13 from ...label import LabeledSentence, SentenceSubcatalog, Word
14 from ..utility import glob
```

(continues on next page)

(continued from previous page)

```

15
16 DATASET_NAME = "THCHS-30"
17 _SEGMENT_NAME_LIST = ("train", "dev", "test")
18
19
20 def THCHS30(path: str) -> Dataset:
21     """Dataloader of the `THCHS-30` dataset.
22
23     .. _THCHS-30: http://166.111.134.19:7777/data/thchs30/README.html
24
25     The file structure should be like::
26
27         <path>
28             lm_word/
29                 lexicon.txt
30             data/
31                 A11_0.wav.trn
32                 ...
33             dev/
34                 A11_101.wav
35                 ...
36             train/
37             test/
38
39     Arguments:
40         path: The root directory of the dataset.
41
42     Returns:
43         Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.
44
45     """
46     dataset = Dataset(DATASET_NAME)
47     dataset.catalog.sentence = _get_subcatalog(os.path.join(path, "lm_word", "lexicon.txt
48     ↪"))
49     for segment_name in _SEGMENT_NAME_LIST:
50         segment = dataset.create_segment(segment_name)
51         for filename in glob(os.path.join(path, segment_name, "*.wav")):
52             data = Data(filename)
53             label_file = os.path.join(path, "data", os.path.basename(filename) + ".trn")
54             data.label.sentence = _get_label(label_file)
55             segment.append(data)
56     return dataset
57
58 def _get_label(label_file: str) -> List[LabeledSentence]:
59     with open(label_file, encoding="utf-8") as fp:
60         labels = ((Word(text=text) for text in texts.split()) for texts in fp)
61         return [LabeledSentence(*labels)]
62
63
64 def _get_subcatalog(lexion_path: str) -> SentenceSubcatalog:
65     subcatalog = SentenceSubcatalog()

```

(continues on next page)

(continued from previous page)

```
66     with open(lexicon_path, encoding="utf-8") as fp:
67         for line in islice(fp, 4, None):
68             subcatalog.append_lexicon(line.strip().split())
69     return subcatalog
```

See *Sentence annotation* for more details.

Note: Since the *THCHS-30 dataloader* above is already included in TensorBay, so it uses relative import. However, the regular import should be used when writing a new dataloader.

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

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 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")
```

1.2.6 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

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.

```
1 {
2   "CLASSIFICATION": {
3     "categories": [
4       { "name": "alt.atheism" },
5       { "name": "comp.graphics" },
6       { "name": "comp.os.ms-windows.misc" },
7       { "name": "comp.sys.ibm.pc.hardware" },
8       { "name": "comp.sys.mac.hardware" },
9       { "name": "comp.windows.x" },
10      { "name": "misc.forsale" },
11      { "name": "rec.autos" },
12      { "name": "rec.motorcycles" },
13      { "name": "rec.sport.baseball" },
14      { "name": "rec.sport.hockey" },
15      { "name": "sci.crypt" },
16      { "name": "sci.electronics" },
17      { "name": "sci.med" },
18      { "name": "sci.space" },
19      { "name": "soc.religion.christian" },
20      { "name": "talk.politics.guns" },
21      { "name": "talk.politics.mideast" },
22      { "name": "talk.politics.misc" },
23      { "name": "talk.religion.misc" }
24    ]
25  }
26 }
```

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

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

Note: The *categories* in *dataset* “20 Newsgroups” have parent-child relationship, and it use “.” to sparate different levels.

Step 2: Write the Dataloader

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

```

1  #!/usr/bin/env python3
2  #
3  # Copyright 2021 Graviti. Licensed under MIT License.
4  #
5  # pylint: disable=invalid-name
6  # pylint: disable=missing-module-docstring
7
8  import os
9
10 from ...dataset import Data, Dataset
11 from ...label import Classification
12 from .._utility import glob
13
14 DATASET_NAME = "Newsgroups20"
15 SEGMENT_DESCRIPTION_DICT = {
16     "20_newsgroups": "Original 20 Newsgroups data set",
17     "20news-bydate-train": (
18         "Training set of the second version of 20 Newsgroups, "
19         "which is sorted by date and has duplicates and some headers removed"
20     ),
21     "20news-bydate-test": (
22         "Test set of the second version of 20 Newsgroups, "
23         "which is sorted by date and has duplicates and some headers removed"
24     ),
25     "20news-18828": (
26         "The third version of 20 Newsgroups, which has duplicates removed "
27         "and includes only 'From' and 'Subject' headers"
28     ),
29 }
30
31
32 def Newsgroups20(path: str) -> Dataset:
33     """Dataloader of the `20 Newsgroups` dataset.
34
35     .. _20 Newsgroups: http://qwone.com/~jason/20Newsgroups/
36
37     The folder structure should be like::
38
39         <path>
40             20news-18828/
41                 alt.atheism/
42                     49960
43                     51060
44                     51119
45                     51120
46                 ...
47             comp.graphics/
48             comp.os.ms-windows.misc/
49             comp.sys.ibm.pc.hardware/

```

(continues on next page)

(continued from previous page)

```

50         comp.sys.mac.hardware/
51         comp.windows.x/
52         misc.forsale/
53         rec.autos/
54         rec.motorcycles/
55         rec.sport.baseball/
56         rec.sport.hockey/
57         sci.crypt/
58         sci.electronics/
59         sci.med/
60         sci.space/
61         soc.religion.christian/
62         talk.politics.guns/
63         talk.politics.mideast/
64         talk.politics.misc/
65         talk.religion.misc/
66     20news-bydate-test/
67     20news-bydate-train/
68     20_newsgroups/
69
70     Arguments:
71         path: The root directory of the dataset.
72
73     Returns:
74         Loaded :class:`~tensorbay.dataset.dataset.Dataset` instance.
75
76     """
77     root_path = os.path.abspath(os.path.expanduser(path))
78     dataset = Dataset(DATASET_NAME)
79     dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))
80
81     for segment_name, segment_description in SEGMENT_DESCRIPTION_DICT.items():
82         segment_path = os.path.join(root_path, segment_name)
83         if not os.path.isdir(segment_path):
84             continue
85
86         segment = dataset.create_segment(segment_name)
87         segment.description = segment_description
88
89         text_paths = glob(os.path.join(segment_path, "*", "*"))
90         for text_path in text_paths:
91             category = os.path.basename(os.path.dirname(text_path))
92
93             data = Data(
94                 text_path, target_remote_path=f"{category}/{os.path.basename(text_path)}.
↪txt"
95             )
96             data.label.classification = Classification(category)
97             segment.append(data)
98
99     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.

Note: Since the *20 Newsgroups dataloader* above is already included in TensorBay, so it uses relative import. However, use regular import should be used when writing a new dataloader.

```
from tensorbay.dataset import Data, Dataset
from tensorbay.label import LabeledBox2D
from tensorbay.opendataset._utility import glob
```

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")
```

1.2.7 Update Dataset

This topic describes how to update datasets, including:

- *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 Label

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

Get a previously uploaded dataset and create a draft:

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

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.delete_segment("SegmentName")
```

Delete data by the file list.

```
segment_client = dataset_client.get_segment("SegmentName")
segment_client.delete_data(["a.png", "b.png"])
```

1.2.8 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)
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()
```

(continues on next page)

(continued from previous page)

```
# 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)
    for file_name in segment_client.list_data_paths():
        if file_name.startswith("Aabyssinian"):
            target_segment_client.copy_data(file_name, file_name, 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())
```

(continues on next page)

(continued from previous page)

```
# 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_random_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_random_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 ResponseSystemError.
 - skip: skip moving or copying segment/data.
 - override: override the whole target segment/data with the source segment/data.
-

1.3 Dataset Management

This topic describes dataset management, including:

- *Organize Dataset*
- *Upload Dataset*
- *Read Dataset*
- *Update Dataset*
- *Move and Copy*

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 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.

Take the *Uploading 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 *Uploading of BSTLD* as an example.

1.3.4 Update Dataset

Since TensorBay supports version control, users can update 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.

1.4 Version Control

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

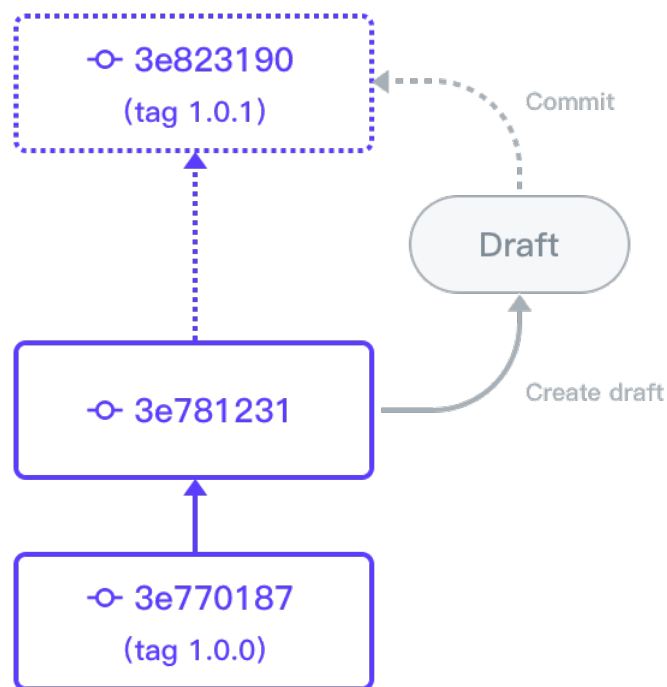


Fig. 1.4: The relations between different versions of a dataset.

1.4.1 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.

```
drafts = dataset_client.list_drafts()
```

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)
```

1.4.2 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")
```

1.4.3 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")
```

1.5 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.

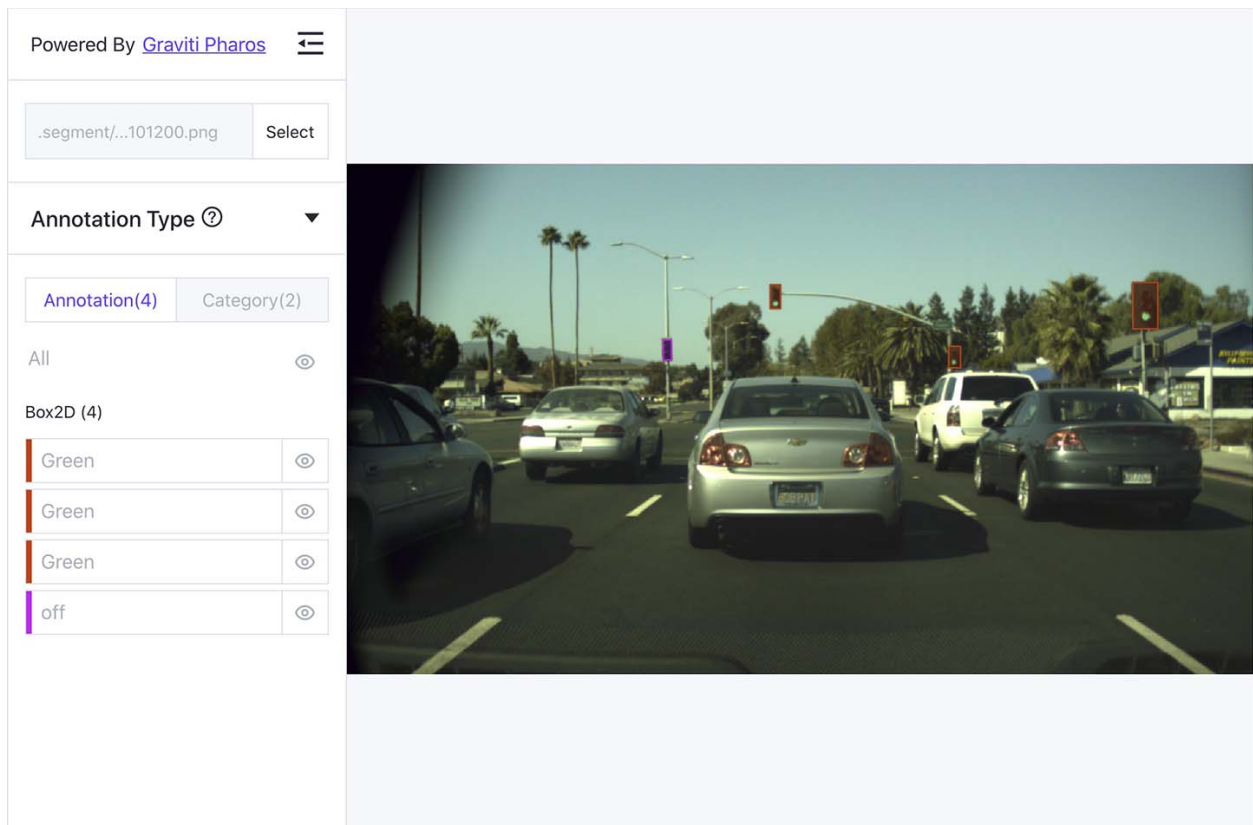


Fig. 1.5: The visualized result of the BSTLD dataset.

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
<i>Camera</i>	image
<i>FisheyeCamera</i>	image
<i>Lidar</i>	point cloud
<i>Radar</i>	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. 1.6). See [this page](#) for more details about this dataset.

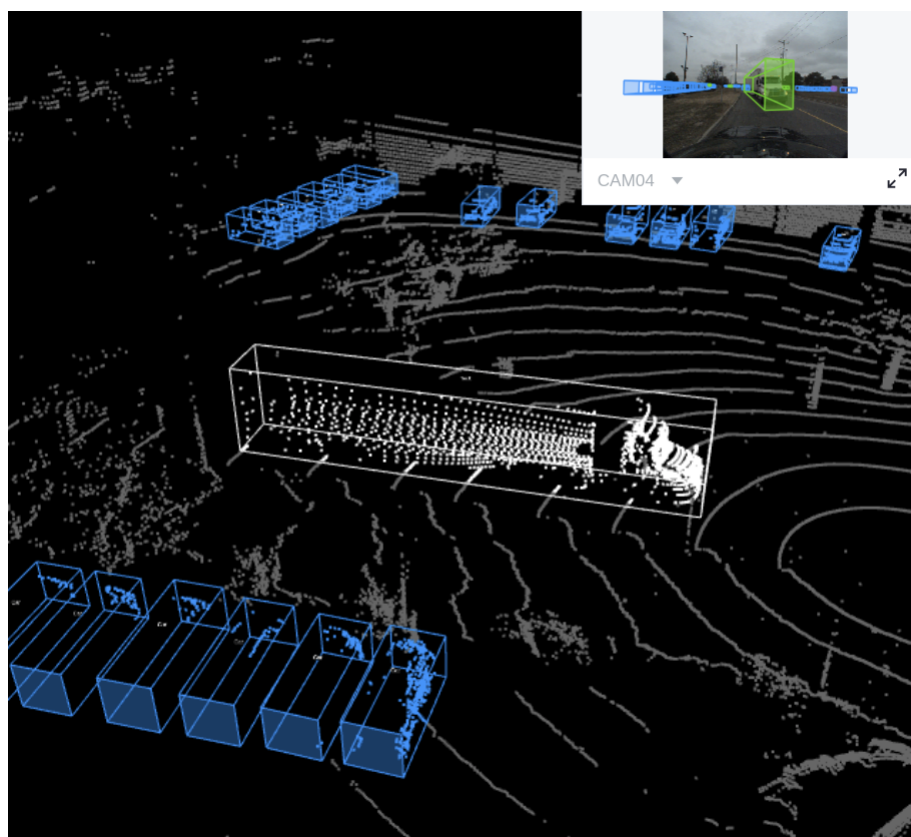


Fig. 1.6: The preview of a point cloud from “CAD-C” with Box3D labels.

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.

```
1 {
2   "BOX3D": {
3     "isTracking": true,
4     "categories": [
5       { "name": "Animal" },
6       { "name": "Bicycle" },
7       { "name": "Bus" },
8       { "name": "Car" },
9       { "name": "Garbage_Container_on_Wheels" },
10      { "name": "Pedestrian" },
11      { "name": "Pedestrian_With_Object" },
12      { "name": "Traffic_Guidance_Objects" },
13      { "name": "Truck" },
14      { "name": "Horse and Buggy" }
15    ],
```

(continues on next page)

(continued from previous page)

```

16     "attributes": [
17         {
18             "name": "stationary",
19             "type": "boolean"
20         },
21         {
22             "name": "camera_used",
23             "enum": [0, 1, 2, 3, 4, 5, 6, 7, null]
24         },
25         {
26             "name": "state",
27             "enum": ["Moving", "Parked", "Stopped"],
28             "parentCategories": ["Car", "Truck", "Bus", "Bicycle", "Horse_and_Buggy"]
29         },
30         {
31             "name": "truck_type",
32             "enum": [
33                 "Construction_Truck",
34                 "Emergency_Truck",
35                 "Garbage_Truck",
36                 "Pickup_Truck",
37                 "Semi_Truck",
38                 "Snowplow_Truck"
39             ],
40             "parentCategories": ["Truck"]
41         },
42         {
43             "name": "bus_type",
44             "enum": ["Coach_Bus", "Transit_Bus", "Standard_School_Bus", "Van_School_
↳ Bus"],
45             "parentCategories": ["Bus"]
46         },
47         {
48             "name": "age",
49             "enum": ["Adult", "Child"],
50             "parentCategories": ["Pedestrian", "Pedestrian_With_Object"]
51         },
52         {
53             "name": "traffic_guidance_type",
54             "enum": ["Permanent", "Moveable"],
55             "parentCategories": ["Traffic_Guidance_Objects"]
56         },
57         {
58             "name": "rider_state",
59             "enum": ["With_Rider", "Without_Rider"],
60             "parentCategories": ["Bicycle"]
61         },
62         {
63             "name": "points_count",
64             "type": "integer",
65             "minimum": 0
66         }

```

(continues on next page)

(continued from previous page)

```

67     ]
68     }
69 }

```

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.

```

1  #!/usr/bin/env python3
2  #
3  # Copyright 2021 Graviti. Licensed under MIT License.
4  #
5  # pylint: disable=invalid-name
6  # pylint: disable=missing-module-docstring
7
8  import json
9  import os
10 from datetime import datetime
11 from typing import Any, Dict, List
12
13 import quaternion
14
15 from ...dataset import Data, Frame, FusionDataset
16 from ...exception import ModuleImportError
17 from ...label import LabeledBox3D
18 from ...sensor import Camera, Lidar, Sensors
19 from ..utility import glob
20
21 DATASET_NAME = "CADC"
22
23
24 def CADC(path: str) -> FusionDataset:
25     """Dataloader of the `CADC` dataset.
26
27     .. _CADC: http://cadcd.uwaterloo.ca/index.html
28
29     The file structure should be like::
30
31         <path>
32         2018_03_06/
33         0001/
34             3d_ann.json
35             labeled/
36                 image_00/
37                     data/
38                         000000000000.png

```

(continues on next page)

(continued from previous page)

```

39         00000000001.png
40         ...
41         timestamps.txt
42         ...
43         image_07/
44             data/
45                 timestamps.txt
46         lidar_points/
47             data/
48                 timestamps.txt
49         novatel/
50             data/
51                 dataformat.txt
52                 timestamps.txt
53         ...
54     0018/
55     calib/
56         00.yaml
57         01.yaml
58         02.yaml
59         03.yaml
60         04.yaml
61         05.yaml
62         06.yaml
63         07.yaml
64         extrinsics.yaml
65         README.txt
66     2018_03_07/
67     2019_02_27/
68
69 Arguments:
70     path: The root directory of the dataset.
71
72 Returns:
73     Loaded `~tensorbay.dataset.dataset.FusionDataset` instance.
74
75 """
76 root_path = os.path.abspath(os.path.expanduser(path))
77
78 dataset = FusionDataset(DATASET_NAME)
79 dataset.notes.is_continuous = True
80 dataset.load_catalog(os.path.join(os.path.dirname(__file__), "catalog.json"))
81
82 for date in os.listdir(root_path):
83     date_path = os.path.join(root_path, date)
84     sensors = _load_sensors(os.path.join(date_path, "calib"))
85     for index in os.listdir(date_path):
86         if index == "calib":
87             continue
88
89     segment = dataset.create_segment(f"{date}-{index}")
90     segment.sensors = sensors

```

(continues on next page)

(continued from previous page)

```

91     segment_path = os.path.join(root_path, date, index)
92     data_path = os.path.join(segment_path, "labeled")
93
94     with open(os.path.join(segment_path, "3d_ann.json"), "r") as fp:
95         # The first line of the json file is the json body.
96         annotations = json.loads(fp.readline())
97         timestamps = _load_timestamps(sensors, data_path)
98         for frame_index, annotation in enumerate(annotations):
99             segment.append(_load_frame(sensors, data_path, frame_index, annotation,
100 ↪ timestamps))
101
102     return dataset
103
104 def _load_timestamps(sensors: Sensors, data_path: str) -> Dict[str, List[str]]:
105     timestamps = {}
106     for sensor_name in sensors.keys():
107         data_folder = f"image_{sensor_name[-2:]}" if sensor_name != "LIDAR" else "lidar_
108 ↪ points"
109         timestamp_file = os.path.join(data_path, data_folder, "timestamps.txt")
110         with open(timestamp_file, "r") as fp:
111             timestamps[sensor_name] = fp.readlines()
112
113     return timestamps
114
115 def _load_frame(
116     sensors: Sensors,
117     data_path: str,
118     frame_index: int,
119     annotation: Dict[str, Any],
120     timestamps: Dict[str, List[str]],
121 ) -> Frame:
122     frame = Frame()
123     for sensor_name in sensors.keys():
124         # The data file name is a string of length 10 with each digit being a number:
125         # 0000000000.jpg
126         # 0000000001.bin
127         data_file_name = f"{frame_index:010}"
128
129         # Each line of the timestamps file looks like:
130         # 2018-03-06 15:02:33.000000000
131         timestamp = datetime.strptime(
132             timestamps[sensor_name][frame_index][:23], "%Y-%m-%d %H:%M:%S.%f"
133         ).timestamp()
134         if sensor_name != "LIDAR":
135             # The image folder corresponds to different cameras, whose name is likes
136 ↪ "CAM00".
137             # The image folder looks like "image_00".
138             camera_folder = f"image_{sensor_name[-2:]}"
139             image_file = f"{data_file_name}.png"

```

(continues on next page)

(continued from previous page)

```

140         data = Data(
141             os.path.join(data_path, camera_folder, "data", image_file),
142             target_remote_path=f"{camera_folder}-{image_file}",
143             timestamp=timestamp,
144         )
145     else:
146         data = Data(
147             os.path.join(data_path, "lidar_points", "data", f"{data_file_name}.bin"),
148             timestamp=timestamp,
149         )
150         data.label.box3d = _load_labels(annotation["cuboids"])
151
152     frame[sensor_name] = data
153     return frame
154
155
156 def _load_labels(boxes: List[Dict[str, Any]]) -> List[LabeledBox3D]:
157     labels = []
158     for box in boxes:
159         dimension = box["dimensions"]
160         position = box["position"]
161
162         attributes = box["attributes"]
163         attributes["stationary"] = box["stationary"]
164         attributes["camera_used"] = box["camera_used"]
165         attributes["points_count"] = box["points_count"]
166
167         label = LabeledBox3D(
168             size=(
169                 dimension["y"], # The "y" dimension is the width from front to back.
170                 dimension["x"], # The "x" dimension is the width from left to right.
171                 dimension["z"],
172             ),
173             translation=(
174                 position["x"], # "x" axis points to the forward facing direction of the
175                 position["y"], # "y" axis points to the left direction of the object.
176                 position["z"],
177             ),
178             rotation=quaternion.from_rotation_vector((0, 0, box["yaw"])),
179             category=box["label"],
180             attributes=attributes,
181             instance=box["uuid"],
182         )
183         labels.append(label)
184
185     return labels
186
187
188 def _load_sensors(calib_path: str) -> Sensors:
189     try:
190         import yaml # pylint: disable=import-outside-toplevel

```

(continues on next page)

(continued from previous page)

```

191     except ModuleNotFoundError as error:
192         raise ModuleImportError(error.name, "pyyaml") from error # type: ignore[arg-
↳ type]
193
194     sensors = Sensors()
195
196     lidar = Lidar("LIDAR")
197     lidar.set_extrinsics()
198     sensors.add(lidar)
199
200     with open(os.path.join(calib_path, "extrinsics.yaml"), "r") as fp:
201         extrinsics = yaml.load(fp, Loader=yaml.FullLoader)
202
203     for camera_calibration_file in glob(os.path.join(calib_path, "[0-9]*.yaml")):
204         with open(camera_calibration_file, "r") as fp:
205             camera_calibration = yaml.load(fp, Loader=yaml.FullLoader)
206
207             # camera_calibration_file looks like:
208             # /path-to-CADC/2018_03_06/calib/00.yaml
209             camera_name = f"CAM{os.path.splitext(os.path.basename(camera_calibration_
↳ file))[0]}"
210             camera = Camera(camera_name)
211             camera.description = camera_calibration["camera_name"]
212
213             camera.set_extrinsics(matrix=extrinsics[f"T_LIDAR_{camera_name}"])
214
215             camera_matrix = camera_calibration["camera_matrix"]["data"]
216             camera.set_camera_matrix(matrix=[camera_matrix[:3], camera_matrix[3:6], camera_
↳ matrix[6:9]])
217
218             distortion = camera_calibration["distortion_coefficients"]["data"]
219             camera.set_distortion_coefficients(**dict(zip(("k1", "k2", "p1", "p2", "k3"),
↳ distortion)))
220
221     sensors.add(camera)
222     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 Cloud Storage

All data on TensorBay are hosted on cloud.

TensorBay supports two cloud storage modes:

- DEFAULT CLOUD STORAGE: data are stored on TensorBay cloud
- AUTHORIZED CLOUD STORAGE: data are stored on other providers’ cloud

1.7.1 Default Cloud Storage

In default cloud storage mode, data are stored on TensorBay cloud.

Create a dataset with default storage:

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

1.7.2 Authorized Cloud Storage

You can also upload data to your public cloud storage space.

Now TensorBay support following cloud providers:

- Aliyun OSS
- Amazon S3
- Azure Blob

Config

See [cloud storage instruction](#) for details about how to configure cloud storage on TensorBay.

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

```
from tensorbay import GAS

gas = GAS("Accesskey-*****")
gas.list_auth_storage_configs()
```

Create Authorized Storage Dataset

Create a dataset with authorized cloud storage:

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

Import Cloud Files into Authorized Storage Dataset

Take the following cloud 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 cloud platform 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")

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

(continues on next page)

(continued from previous page)

```
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 raw directory to the authorized cloud storage dataset path, thus the storage space will be doubled on the cloud platform.

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 <code>allowed_retry_methods</code> and the response status is one of the <code>allowed_retry_status</code> , then the request can auto-retry <i>max_retries</i> 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.

```

(continues on next page)

(continued from previous page)

```
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()
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 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):
        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}")
```

1.11 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.
            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)
```

(continues on next page)

(continued from previous page)

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

1.12 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 can:

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

1.12.1 Installation

To use TensorBay CLI, please install TensorBay SDK first.

```
$ pip3 install tensorbay
```

1.12.2 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
```

1.12.3 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>]
```

1.12.4 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>]
```

1.12.5 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
```

1.13 TensorBay Resource Name

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

1. Define a dataset

```
tb:<dataset_name>
```

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

```
tb:VOC2012
```

2. Define a segment

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

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

```
tb:VOC2010:train
```

3. Define 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
```

1.13.1 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.

1.14 CLI Commands

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

Table 1.4: CLI Commands

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

1.14.1 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]
```

1.14.2 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
```

1.14.3 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>
```

1.14.4 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>
```

1.14.5 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>
```

1.14.6 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>
```

1.14.7 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>
```

1.14.8 gas commit

Work with commit operations.

Commit a *draft* with a title.

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

1.14.9 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>
```

1.14.10 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>
```

1.14.11 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>
```

1.15 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.
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.

1.15.1 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)
```

1.15.2 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.

1.16 Glossary

1.16.1 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 configuration](#).

1.16.2 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.

1.16.3 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.

1.16.4 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.

1.16.5 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 <i>Box2D</i> label.
Dogs vs Cats Dataloader	This example is the dataloader of Dogs vs Cats Dataset, which is a dataset with <i>Classification</i> label.
BSTLD Dataloader	This example is the dataloader of BSTLD Dataset, which is a dataset with <i>Box2D</i> label.
Neolix OD Dataloader	This example is the dataloader of Neolix OD Dataset, which is a dataset with <i>Box3D</i> label.
Leeds Sports Pose Daraloader	This example is the dataloader of Leeds Sports Pose Dataset, which is a dataset with <i>Keypoints2D</i> 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*.

1.16.6 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.

1.16.7 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.

1.16.8 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.

1.16.9 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*.

1.16.10 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).

1.16.11 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
```

1.16.12 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.

1.17 Dataset Structure

For ease of use, TensorBay defines a uniform dataset format. This topic explains the related concepts. The TensorBay dataset format looks like:



1.17.1 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*.

1.17.2 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*.

1.17.3 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 <i>Classification</i> label.
BSTLD Catalog	This example is the catalog of BSTLD Dataset, which is a dataset with <i>Box2D</i> label.
Neolix OD Catalog	This example is the catalog of Neolix OD Dataset, which is a dataset with <i>Box3D</i> label.
Leeds Sports Pose Catalog	This example is the catalog of Leeds Sports Pose Dataset, which is a dataset with <i>Keypoints2D</i> label.
NightOwls Catalog	This example is the catalog of NightOwls Dataset, which is a dataset with tracking <i>Box2D</i> label.

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

1.17.4 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*.

1.17.5 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*.

1.18 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
<i>Classification</i>	<i>Classification</i>	<i>ClassificationSubcatalog</i>
<i>Box2D</i>	<i>LabeledBox2D</i>	<i>Box2DSubcatalog</i>
<i>Box3D</i>	<i>LabeledBox3D</i>	<i>Box3DSubcatalog</i>
<i>Keypoints2D</i>	<i>LabeledKeypoints2D</i>	<i>Keypoints2DSubcatalog</i>
<i>Polygon</i>	<i>LabeledPolygon</i>	<i>PolygonSubcatalog</i>
<i>MultiPolygon</i>	<i>LabeledMultiPolygon</i>	<i>MultiPolygonSubcatalog</i>
<i>RLE</i>	<i>LabeledRLE</i>	<i>RLESubcatalog</i>
<i>Polyline2D</i>	<i>LabeledPolyline2D</i>	<i>Polyline2DSubcatalog</i>
<i>MultiPolyline2D</i>	<i>LabeledMultiPolyline2D</i>	<i>MultiPolyline2DSubcatalog</i>
<i>Sentence</i>	<i>LabeledSentence</i>	<i>SentenceSubcatalog</i>
<i>SemanticMask</i>	<i>SemanticMask</i>	<i>SemanticMaskSubcatalog</i>
<i>InstanceMask</i>	<i>InstanceMask</i>	<i>InstanceMaskSubcatalog</i>
<i>PanopticMask</i>	<i>PanopticMask</i>	<i>PanopticMaskSubcatalog</i>

1.18.1 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"
```

1.18.2 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 can 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.

1.18.3 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 apply 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.

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.

1.18.4 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>
    ...
  }
}
```

(continues on next page)

(continued from previous page)

```

    ...
    },
    "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.

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.

1.18.5 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>
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    },
    "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),

```

(continues on next page)

(continued from previous page)

```
)
>>> 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)
)
>>> 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.

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.

1.18.6 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"
... )
```

(continues on next page)

(continued from previous page)

```
>>> keypoints2d_label
LabeledKeypoints2D [
  Keypoint2D(10, 20),
  Keypoint2D(15, 25),
  Keypoint2D(20, 30)
](
  (category): 'category',
  (attributes): {...},
  (instance): 'instance_ID'
)
```

Keypoints2D.keypoints2d

LabeledKeypoints2D extends *Keypoints2D*.

To construct a *LabeledKeypoints2D* instance with only the geometry information, 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.

```
>>> 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.

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

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

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

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

See details in *Keypoint2D*.

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

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

To add a *LabeledKeypoints2D* label to one data:

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

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

1.18.7 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.

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.

1.18.8 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)]],
...   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.

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.

1.18.9 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>
    ...
    ...
  }
  "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,
```

(continues on next page)

(continued from previous page)

```
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.

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.

1.18.10 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>
    },
    ...
  ]
}
```

(continues on next page)

(continued from previous page)

```

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

```

To create a *LabeledPolyline2D* label:

```

>>> from tensorbay.label import LabeledPolyline2D
>>> polyline2d_label = LabeledPolyline2D(
...     [(1, 2), (2, 3)],
...     category="category",
...     attributes={"attribute_name": "attribute_value"},
...     instance="instance_ID"
... )
>>> polyline2d_label
LabeledPolyline2D [
    Vector2D(1, 2),
    Vector2D(2, 3)
](
    (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.

Polyline2DSubcatalog has categories, attributes and tracking information, see [common category information](#), [attributes information](#) and [tracking information](#) for details.

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.

1.18.11 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>
      },
      ...
    ],
    ...
  ],
  "category": <str>
```

(continues on next page)

(continued from previous page)

```
"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.

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.

1.18.12 Sentence

Sentence label is the transcribed 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>
    }
    ...
  ]
}
```

(continues on next page)

(continued from previous page)

```

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

```

To create a *LabeledSentence* label:

```

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

```

(continues on next page)

(continued from previous page)

}

Sentence.sentence

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

Sentence.spell

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

It is only for Chinese language.

Sentence.phone

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

Word

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

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

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

Sentence.attributes

The attributes of the transcribed sentence. See *attributes information* 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 transcribed sentences of the audio.

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

(continues on next page)

(continued from previous page)

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

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

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

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

Besides giving the parameters while initialing *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.

1.18.13 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>  
                ...  
            }  
        }  
    ]  
}
```

(continues on next page)

(continued from previous page)

```

        ...
    }
    },
    ...
    ...
]
}

```

`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
    },
    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.

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.

1.18.14 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
    }
  }
)
```

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.

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.

1.18.15 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": {
```

(continues on next page)

(continued from previous page)

```

        <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
    },

```

(continues on next page)

(continued from previous page)

```

    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.

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.

1.19 Exceptions

TensorBay SDK defines a series of custom exceptions.

TensorBayException *TensorBayException* is the base class for TensorBay SDK custom exceptions.

TBRNError *TBRNError* defines the exception for invalid TBRN. Raised when the TBRN format is incorrect.

ClientError *ClientError* is the base class for custom exceptions in the client module.

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.

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.

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.

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.

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.

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.

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.

ResponseSystemError *ResponseSystemError* defines the exception for system response error in the client module. Raised when system error was responded.

UnauthorizedError *UnauthorizedError* defines the exception for unauthorized response error in the client module. Raised when the *accesskey* is incorrect.

OpenDatasetError *OpenDatasetError* is the base class for custom exceptions in the opendataset module.

NoFileError *NoFileError* defines the exception for no matching file found in the opendataset directory.

FileStructureError *FileStructureError* defines the exception for incorrect file structure in the opendataset directory.

1.19.1 Exception hierarchy

The class hierarchy for TensorBay custom exceptions is:

```
+-- TensorBayException
  +-- ClientError
    +-- StatusError
    +-- DatasetTypeError
    +-- FrameError
    +-- ResponseError
      +-- AccessDeniedError
      +-- InvalidParamsError
      +-- NameConflictError
      +-- RequestParamsMissingError
      +-- ResourceNotExistError
      +-- ResponseSystemError
      +-- UnauthorizedError
  +-- TBRNError
  +-- OpenDatasetError
    +-- NoFileError
    +-- FileStructureError
```

1.20 API Reference

1.20.1 tensorbay.client

tensorbay.client.cloud_storage

Class *CloudClient*.

The *CloudClient* defines the initial client to interact between local and cloud platform.

```
class tensorbay.client.cloud_storage.CloudClient(name: str, client: tensorbay.client.requests.Client)
    Bases: object
```

CloudClient defines the client to interact between local and cloud platform.

Parameters

- **name** – Name of the auth cloud storage config.
- **client** – The initial client to interact between local and TensorBay.

```
list_auth_data(path: str = "") → List[tensorbay.dataset.data.AuthData]
    List all cloud files in the given directory as AuthData.
```

Parameters path – The directory path on the cloud platform.

Returns The list of *AuthData* of all the cloud files.

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: str, dataset_id: str, gas: GAS, *, status:
                                                    tensorbay.client.status.Status, alias: str = "")
```

Bases: *tensorbay.client.version.VersionControlClient*

This class defines the basic concept of the dataset client.

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

Parameters

- **name** – Dataset name.
- **dataset_id** – Dataset ID.
- **gas** – The initial client to interact between local and TensorBay.
- **status** – The version control status of the dataset.
- **alias** – Dataset alias.

name

Dataset name.

dataset_id

Dataset ID.

status

The version control status of the dataset.

alias

Dataset alias.

update_notes(**, is_continuous: Optional[bool] = None, bin_point_cloud_fields: Optional[Iterable[str]] = Ellipsis*) → None

Update the notes.

Parameters

- **is_continuous** – Whether the data is continuous.
- **bin_point_cloud_fields** – The field names of the bin point cloud files in the dataset.

get_notes() → *tensorbay.dataset.dataset.Notes*

Get the notes.

Returns The *Notes*.

list_segment_names() → *tensorbay.client.lazy.PagingList*[str]

List all segment names in a certain commit.

Returns The *PagingList* of segment names.

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

Get the catalog of the certain commit.

Returns Required *Catalog*.

upload_catalog(*catalog: tensorbay.label.catalog.Catalog*) → None

Upload a catalog to the draft.

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

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

delete_segment(*name: str*) → None

Delete a segment of the draft.

Parameters **name** – Segment name.

class *tensorbay.client.dataset.DatasetClient*(*name: str, dataset_id: str, gas: GAS, *, status: tensorbay.client.status.Status, alias: str = ""*)

Bases: *tensorbay.client.dataset.DatasetClientBase*

This class defines *DatasetClient*.

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

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

Get or create a segment with the given name.

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

Returns The created *SegmentClient* with given name.

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

Create a segment with the given name.

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

Returns The created *SegmentClient* with given name.

Raises *NameConflictError* – When the segment exists.

copy_segment(*source_name: str, target_name: Optional[str] = None, *, source_client: Optional[[tensorbay.client.dataset.DatasetClient](#)] = None, strategy: str = 'abort') → [tensorbay.client.segment.SegmentClient](#)*

Copy segment to this dataset.

Parameters

- **source_name** – The source name of the copied segment.
- **target_name** – 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** – 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** – 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.

move_segment(*source_name: str, target_name: str, *, strategy: str = 'abort') → [tensorbay.client.segment.SegmentClient](#)*

Move/Rename segment in this dataset.

Parameters

- **source_name** – The source name of the moved segment.
- **target_name** – The target name of the moved segment.
- **strategy** – 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.

get_segment(*name: str = 'default') → [tensorbay.client.segment.SegmentClient](#)*

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

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

Returns The required [SegmentClient](#).

Raises [ResourceNotExistError](#) – When the required segment does not exist.

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

Upload a [Segment](#) to the dataset.

This function will upload all info contains in the input [Segment](#), which includes:

- Create a segment using the name of input Segment.
- Upload all Data in the Segment to the dataset.

Parameters

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

Raises Exception – When the upload got interrupted by Exception.

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

```
class tensorbay.client.dataset.FusionDatasetClient(name: str, dataset_id: str, gas: GAS, *, status:
                                                    tensorbay.client.status.Status, alias: str = "")
```

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.

```
get_or_create_segment(name: str = 'default') → tensorbay.client.segment.FusionSegmentClient
```

Get or create a fusion segment with the given name.

Parameters name – The name of the fusion segment.

Returns The created *FusionSegmentClient* with given name.

```
create_segment(name: str = 'default') → tensorbay.client.segment.FusionSegmentClient
```

Create a fusion segment with the given name.

Parameters name – The name of the fusion segment.

Returns The created *FusionSegmentClient* with given name.

Raises NameConflictError – When the segment exists.

```
copy_segment(source_name: str, target_name: Optional[str] = None, *, source_client:
              Optional[tensorbay.client.dataset.FusionDatasetClient] = None, strategy: str = 'abort') →
              tensorbay.client.segment.FusionSegmentClient
```

Copy segment to this dataset.

Parameters

- **source_name** – The source name of the copied segment.
- **target_name** – 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** – 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** – 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.

move_segment(*source_name: str, target_name: str, *, strategy: str = 'abort'*) → *tensorbay.client.segment.FusionSegmentClient*

Move/Rename segment in this dataset.

Parameters

- **source_name** – The source name of the moved segment.
- **target_name** – The target name of the moved segment.
- **strategy** – 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.

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

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

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

Returns The required *FusionSegmentClient*.

Raises *ResourceNotExistError* – When the required fusion segment does not exist.

upload_segment(*segment: tensorbay.dataset.segment.FusionSegment, *, jobs: int = 1, skip_uploaded_files: bool = False, quiet: bool = False*) → *tensorbay.client.segment.FusionSegmentClient*

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** – The *FusionSegment*.
- **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.

Raises **Exception** – When the upload got interrupted by Exception.

Returns

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

tensorbay.client.gas

Class GAS.

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

AccessKey is required when operating with dataset.

```
class tensorbay.client.gas.GAS(access_key: str, url: str = "")
```

Bases: object

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

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

Parameters

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

get_user() → *tensorbay.client.struct.UserInfo*

Get the user information with the current accesskey.

Returns The *struct.UserInfo* with the current accesskey.

get_auth_storage_config(name: str) → Dict[str, Any]

Get the auth storage config with the given name.

Parameters **name** – 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.

list_auth_storage_configs() → *tensorbay.client.lazy.PagingList*[Dict[str, Any]]

List auth storage configs.

Returns The *PagingList* of all auth storage configs.

get_cloud_client(name: str) → *tensorbay.client.cloud_storage.CloudClient*

Get a cloud client used for interacting with cloud platform.

Parameters **name** – The required auth storage config name.

Returns The cloud client of this dataset.

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

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

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

Create a TensorBay dataset with given name.

Parameters

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

- **is_fusion** – Whether the dataset is a fusion dataset, True for fusion dataset.
- **region** – Region of the dataset to be stored, only support “beijing”, “hangzhou”, “shanghai”, default is “shanghai”.
- **alias** – Alias of the dataset, default is “”.

Returns The created `DatasetClient` instance or `FusionDatasetClient` instance (is_fusion=True), and the status of dataset client is “commit”.

create_auth_dataset(name: str, config_name: str, *, is_fusion: bool = False, alias: str = "") → Union[`tensorbay.client.dataset.DatasetClient`, `tensorbay.client.dataset.FusionDatasetClient`]

Create a TensorBay dataset with given name in auth cloud storage.

The dataset will be linked to the given auth cloud storage and all of relative data will be stored in auth cloud storage.

Parameters

- **name** – Name of the dataset, unique for a user.
- **config_name** – The auth storage config name.
- **is_fusion** – Whether the dataset is a fusion dataset, True for fusion dataset.
- **alias** – Alias of the dataset, default is “”.

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() → `tensorbay.client.lazy.PagingList`[str]
List names of all TensorBay datasets.

Returns The `PagingList` of all TensorBay dataset names.

update_dataset(name: str, *, alias: Optional[str] = None) → None
Update a TensorBay Dataset.

Parameters

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

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

Rename a TensorBay Dataset with given name.

Parameters

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

upload_dataset(*dataset: tensorbay.dataset.dataset.Dataset, draft_number: Optional[int] = None, *, 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

- **OperationError** – 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: str*) → None

Delete a TensorBay dataset with given name.

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

tensorbay.client.lazy

Lazy evaluation related classes.

```
class tensorbay.client.lazy.LazyItem(page: tensorbay.client.lazy.LazyPage[tensorbay.client.lazy._T],
                                     data: tensorbay.client.lazy._T)
```

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 element, 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: tensorbay.client.lazy.LazyPage[tensorbay.client.lazy._T]) →
                    tensorbay.client.lazy.LazyItem[tensorbay.client.lazy._T]
```

Create a LazyItem instance from page.

Parameters **page** – The page of the element.

Returns The LazyItem instance which stores the input page.

```
classmethod from_data(data: tensorbay.client.lazy._T) →
                    tensorbay.client.lazy.LazyItem[tensorbay.client.lazy._T]
```

Create a LazyItem instance from data.

Parameters **data** – The actual data needs to be stored in LazyItem.

Returns The LazyItem instance which stores the input data.

```
get() → tensorbay.client.lazy._T
```

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 element.

Returns The actual element this LazyItem instance represents.

```
class tensorbay.client.lazy.ReturnGenerator(generator: Generator[tensorbay.client.lazy._T, Any,
                                                                tensorbay.client.lazy._R])
```

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: int, limit: int, func: Callable[[int, int],
                                                                Generator[tensorbay.client.lazy._T, None, int]])
```

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.

pull() → None

Send paging request to pull a page of elements and store them in [LazyItem](#).

```
class tensorbay.client.lazy.InitPage(offset: int, limit: int, func: Callable[[int, int],
                                Generator[tensorbay.client.lazy._T, None, int]])
```

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: Callable[[int, int], Generator[tensorbay.client.lazy._T,
                                None, int]], limit: int)
```

Bases: [MutableSequence](#)[[tensorbay.client.lazy._T](#)], [tensorbay.utility.repr.ReprMixin](#)

PagingList is a wrap of web paging request.

It follows the python MutableSequence protocol, 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: int, value: tensorbay.client.lazy._T) → None

Insert object before index.

Parameters

- **index** – Position of the PagingList.
- **value** – Element to be inserted into the PagingList.

append(*value: tensorbay.client.lazy._T*) → None
Append object to the end of the PagingList.

Parameters value – Element to be appended to the PagingList.

reverse() → None
Reverse the items of the PagingList in place.

pop(*index: int = -1*) → *tensorbay.client.lazy._T*
Return the item at index (default last) and remove it from the PagingList.

Parameters index – Position of the PagingList.

Returns Element to be removed from the PagingList.

index(*value: Any, start: int = 0, stop: Optional[int] = None*) → int
Return the first index of the value.

Parameters

- **value** – The value to be found.
- **start** – The start index of the subsequence.
- **stop** – The end index of the subsequence.

Raises ValueError – When the value is not in the PagingList

Returns The first index of the value.

count(*value: Any*) → int
Return the number of occurrences of value.

Parameters value – The value needs to be counted.

Returns The number of occurrences of value.

extend(*values: Iterable[tensorbay.client.lazy._T]*) → None
Extend PagingList by appending elements from the iterable.

Parameters values – Elements to be extended into the PagingList.

tensorbay.client.log

Logging utility functions.

Dump_request_and_response dumps http request and response.

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

This class used to lazy load request to logging.

Parameters request – The request of the request.

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

This class used to lazy load response to logging.

Parameters response – The response of the request.

`tensorbay.client.log.dump_request_and_response(response: requests.models.Response) → str`
 Dumps http request and response.

Parameters `response` – Http response and response.

Returns

Http request and response for logging, sample:

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

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

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

--5adff1fc0524465593d6a9ad68aad7f9--

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

(continues on next page)

(continued from previous page)

```

"code": "DATACENTER-0",
"message": "success",
"data": {}
}
=====

```

tensorbay.client.requests

Class Client and method multithread_upload.

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

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

class tensorbay.client.requests.**Config**

Bases: object

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

max_retries

Maximum retry times of the request.

allowed_retry_methods

The allowed methods for retrying request.

allowed_retry_status

The allowed status for retrying request. If both methods and status are fitted, the retrying strategy will work.

timeout

Timeout value of the request in seconds.

is_internal

Whether the request is from internal.

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

Bases: requests.adapters.HTTPAdapter

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

Parameters

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

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

Parameters

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

- **cert** – User-provided SSL certificate.
- **proxies** – Proxies dict applying to the request.

Returns Response object.

class `tensorbay.client.requests.UserSession`

Bases: `requests.sessions.Session`

This class defines `UserSession`.

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

Parameters

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

Returns Response of the request.

Raises [`ResponseError`](#) – If post response error.

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

Bases: `object`

This class defines [`Client`](#).

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

Parameters

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

property session: `tensorbay.client.requests.UserSession`

Create and return a session per PID so each sub-processes will use their own session.

Returns The session corresponding to the process.

open_api_do(*method: str, section: str, dataset_id: str = "", **kwargs: Any*) → `requests.models.Response`
Send a request to the TensorBay Open API.

Parameters

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

Raises [`ResponseError`](#) – When the status code OpenAPI returns is unexpected.

Returns Response of the request.

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

Parameters

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

Returns Response of the request.

class `tensorbay.client.requests.Tqdm(*_, **__)`

Bases: `tqdm.std.tqdm`

A wrapper class of `tqdm` for showing the process bar.

Parameters

- **total** – The number of excepted iterations.
- **disable** – Whether to disable the entire progress bar.

update_callback(_: *Any*) → *None*

Callback function for updating process bar when multithread task is done.

update_for_skip(*condition: bool*) → *bool*

Update process bar for the items which are skipped in builtin filter function.

Parameters **condition** – The filter condition, the process bar will be updated if condition is *False*.

Returns The input condition.

`tensorbay.client.requests.multithread_upload(function: Callable[[tensorbay.client.requests._T], Optional[tensorbay.client.requests._R]], arguments: Iterable[tensorbay.client.requests._T], *, callback: Optional[Callable[[Tuple[tensorbay.client.requests._R, ...]], None]] = None, jobs: int = 1, pbar: tensorbay.client.requests.Tqdm) → None`

Multi-thread upload framework.

Parameters

- **function** – The upload function.
- **arguments** – The arguments of the upload function.
- **callback** – The callback function.
- **jobs** – The number of the max workers in multi-thread uploading procession.
- **pbar** – The [Tqdm](#) instance for showing the upload process bar.

class `tensorbay.client.requests.MultiCallbackTask(*, function: Callable[[tensorbay.client.requests._T], Optional[tensorbay.client.requests._R]], callback: Callable[[Tuple[tensorbay.client.requests._R, ...]], None], size: int = 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: tensorbay.client.requests._T*) → None

Do the work of a single thread.

Parameters **argument** – The argument of the function.

last_callback() → None

Send the last callback when all works have been done.

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: str, dataset_client: Union[DatasetClient, FusionDatasetClient]*)

Bases: object

This class defines the basic concept of *SegmentClient*.

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

Parameters

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

name

Segment name.

status

The status of the dataset client.

delete_data(*remote_paths: Union[str, Iterable[str]]*) → None

Delete data of a segment in a certain commit with the given remote paths.

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

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

Bases: *tensorbay.client.segment.SegmentClientBase*

This class defines *SegmentClient*.

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

upload_file(*local_path: str, target_remote_path: str = ""*) → None

Upload data with local path to the draft.

Parameters

- **local_path** – The local path of the data to upload.

- **target_remote_path** – The path to save the data in segment client.

upload_label(data: [tensorbay.dataset.data.Data](#)) → None

Upload label with Data object to the draft.

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

upload_data(data: [tensorbay.dataset.data.Data](#)) → None

Upload Data object to the draft.

Parameters data – The [Data](#).

import_auth_data(data: [tensorbay.dataset.data.AuthData](#)) → None

Import AuthData object to the draft.

Parameters data – The [Data](#).

copy_data(source_remote_paths: Union[str, Iterable[str]], target_remote_paths: Union[None, str, Iterable[str]] = None, *, source_client: Optional[[tensorbay.client.segment.SegmentClient](#)] = None, strategy: str = 'abort') → None

Copy data to this segment.

Parameters

- **source_remote_paths** – The source remote paths of the copied data.
- **target_remote_paths** – 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** – 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** – 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.
- [OperationError](#) – When the type of target_remote_paths is not equal with source_remote_paths.

move_data(source_remote_paths: Union[str, Iterable[str]], target_remote_paths: Union[None, str, Iterable[str]] = None, *, source_client: Optional[[tensorbay.client.segment.SegmentClient](#)] = None, strategy: str = 'abort') → None

Move data to this segment, also used to rename data.

Parameters

- **source_remote_paths** – The source remote paths of the moved data.
- **target_remote_paths** – 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** – 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** – 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.
- **OperationError** – 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.

list_data_paths() → *tensorbay.client.lazy.PagingList*[str]

List required data path in a segment in a certain commit.

Returns The *PagingList* of data paths.

list_data() → *tensorbay.client.lazy.PagingList*[*tensorbay.dataset.data.RemoteData*]

List required Data object in a dataset segment.

Returns The *PagingList* of *RemoteData*.

list_urls() → *tensorbay.client.lazy.PagingList*[str]

List the data urls in this segment.

Returns The *PagingList* of urls.

list_mask_urls(mask_type: str) → *tensorbay.client.lazy.PagingList*[str]

List the mask urls in this segment.

Parameters mask_type – The required mask type, the supported types are SEMANTIC_MASK, INSTANCE_MASK and PANOPTIC_MASK

Returns The *PagingList* of mask urls.

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

Bases: *tensorbay.client.segment.SegmentClientBase*

This class defines *FusionSegmentClient*.

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

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

get_sensors() → *tensorbay.sensor.sensor.Sensors*

Return the sensors in a fusion segment client.

Returns The sensors in the fusion segment client.

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

Upload sensor to the draft.

Parameters sensor – The sensor to upload.

delete_sensor(sensor_name: str) → None

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

Parameters sensor_name – The TensorBay sensor to delete.

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

Upload frame to the draft.

Parameters

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

Raises *FrameError* – When lacking frame id or frame id conflicts.

list_frames() → *tensorbay.client.lazy.PagingList[tensorbay.dataset.frame.Frame]*

List required frames in the segment in a certain commit.

Returns The PagingList of *Frame*.

list_urls() → *tensorbay.client.lazy.PagingList[Dict[str, str]]*

List the data urls in this segment.

Returns The PagingList of url dict, which key is the sensor name, value is the url.

tensorbay.client.status

Class Status.

class *tensorbay.client.status.Status*(*branch_name: Optional[str] = None, *, draft_number: Optional[int] = None, commit_id: Optional[str] = None*)

Bases: object

This class defines the basic concept of the status.

Parameters

- **branch_name** – The branch name.
- **draft_number** – The draft number (if the status is draft).
- **commit_id** – The commit ID (if the status is commit).

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() → Dict[str, Any]

Get the dict containing the draft number or commit ID.

Returns A dict containing the draft number or commit ID.

check_authority_for_commit() → None

Check whether the status is a legal commit.

Raises *StatusError* – When the status is not a legal commit.

check_authority_for_draft() → None

Check whether the status is a legal draft.

Raises *StatusError* – When the status is not a legal draft.

checkout (*commit_id: Optional[str] = None, draft_number: Optional[int] = None*) → None
Checkout to commit or draft.

Parameters

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

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: str, *, email: Optional[str] = None, description: str = ""*)
Bases: *tensorbay.utility.name.NameMixin*

This class defines the basic concept of a TensorBay team.

Parameters

- **name** – The name of the team.
- **email** – The email of the team.
- **description** – The description of the team.

classmethod **loads**(*contents: Dict[str, Any]*) → tensorbay.client.struct._T
Loads a *TeamInfo* instance from the given contents.

Parameters **contents** – 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.

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

Returns

A dict containing all the information of the team:

```
{
    "name": <str>
    "email": <str>
    "description": <str>
}
```

```
class tensorbay.client.struct.UserInfo(name: str, *, email: Optional[str] = None, mobile: Optional[str]
                                     = None, description: str = "", team:
                                     Optional[tensorbay.client.struct.TeamInfo] = None)
```

Bases: [tensorbay.utility.name.NameMixin](#)

This class defines the basic concept of a TensorBay user.

Parameters

- **name** – The nickname of the user.
- **email** – The email of the user.
- **mobile** – The mobile of the user.
- **description** – The description of the user.
- **team** – The team of the user.

```
classmethod loads(contents: Dict[str, Any]) → tensorbay.client.struct._T
```

Loads a [UserInfo](#) instance from the given contents.

Parameters contents – 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.

```
dumps() → Dict[str, Any]
```

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>
    }
}
```

```
class tensorbay.client.struct.User(name: str, date: int)
```

Bases: [tensorbay.utility.attr.AttrsMixin](#), [tensorbay.utility.repr.ReprMixin](#)

This class defines the basic concept of a user with an action.

Parameters

- **name** – The name of the user.
- **date** – The date of the user action.

classmethod `loads(contents: Dict[str, Any]) → tensorbay.client.struct._T`

Loads a `User` instance from the given contents.

Parameters **contents** – 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.

dumps() → Dict[str, Any]

Dumps all the user information into a dict.

Returns

A dict containing all the information of the user:

```
{
    "name": <str>
    "date": <int>
}
```

class `tensorbay.client.struct.Commit(commit_id: str, parent_commit_id: Optional[str], title: str, description: str, committer: tensorbay.client.struct.User)`

Bases: `tensorbay.utility.attr.AttrsMixin`, `tensorbay.utility.repr.ReprMixin`

This class defines the structure of a commit.

Parameters

- **commit_id** – The commit id.
- **parent_commit_id** – The parent commit id.
- **title** – The commit title.
- **description** – The commit description.
- **committer** – The commit user.

classmethod `loads(contents: Dict[str, Any]) → tensorbay.client.struct._T`

Loads a `Commit` instance for the given contents.

Parameters **contents** – A dict containing all the information of the commit:

```
{
    "commitId": <str>
    "parentCommitId": <str> or None
    "title": <str>
    "description": <str>
    "committer": {
        "name": <str>
        "date": <int>
    }
}
```

(continues on next page)

(continued from previous page)

```
}
}
```

Returns A *Commit* instance containing all the information in the given contents.

dumps() → Dict[str, Any]

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>
  }
}
```

class tensorbay.client.struct.**Tag**(name: str, commit_id: str, parent_commit_id: Optional[str], title: str, description: str, committer: tensorbay.client.struct.User)

Bases: tensorbay.client.struct._NamedCommit

This class defines the structure of the tag of a commit.

Parameters

- **name** – The name of the tag.
- **commit_id** – The commit id.
- **parent_commit_id** – The parent commit id.
- **title** – The commit title.
- **description** – The commit description.
- **committer** – The commit user.

class tensorbay.client.struct.**Branch**(name: str, commit_id: str, parent_commit_id: Optional[str], title: str, description: str, committer: tensorbay.client.struct.User)

Bases: tensorbay.client.struct._NamedCommit

This class defines the structure of a branch.

Parameters

- **name** – The name of the branch.
- **commit_id** – The commit id.
- **parent_commit_id** – The parent commit id.
- **title** – The commit title.
- **description** – The commit description.
- **committer** – The commit user.

```
class tensorbay.client.struct.Draft(number: int, title: str, branch_name: str, status: str, description: str = "")
```

Bases: `tensorbay.utility.attr.AttrsMixin`, `tensorbay.utility.repr.ReprMixin`

This class defines the basic structure of a draft.

Parameters

- **number** – The number of the draft.
- **title** – The title of the draft.
- **branch_name** – The branch name.
- **status** – The status of the draft.
- **description** – The draft description.

```
classmethod loads(contents: Dict[str, Any]) → tensorbay.client.struct._T
```

Loads a `Draft` instance from the given contents.

Parameters **contents** – A dict containing all the information of the draft:

```
{
    "number": <int>
    "title": <str>
    "branchName": <str>
    "status": "OPEN", "CLOSED" or "COMMITTED"
    "description": <str>
}
```

Returns A `Draft` instance containing all the information in the given contents.

```
dumps() → Dict[str, Any]
```

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"
    "description": <str>
}
```

tensorbay.client.version

TensorBay dataset version control related classes.

```
class tensorbay.client.version.VersionControlClient(dataset_id: str, gas: GAS, *, status: tensorbay.client.status.Status)
```

Bases: `object`

TensorBay dataset version control client.

Parameters

- **dataset_id** – Dataset ID.

- **gas** – The initial client to interact between local and TensorBay.
- **status** – The version control status of the dataset.

property dataset_id: `str`

Return the TensorBay dataset ID.

Returns The TensorBay dataset ID.

property status: `tensorbay.client.status.Status`

Return the status of the dataset client.

Returns The status of the dataset client.

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

Checkout to commit or draft.

Parameters

- **revision** – The information to locate the specific commit, which can be the commit id, the branch, or the tag.
- **draft_number** – The draft number.

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

commit(*title: str, description: str = "", *, tag: Optional[str] = None*) → 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** – The commit title.
- **description** – The commit description.
- **tag** – A tag for current commit.

create_draft(*title: str, description: str = "", branch_name: Optional[str] = None*) → int

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** – The draft title.
- **description** – The draft description.
- **branch_name** – The branch name.

Returns The draft number of the created draft.

Raises **StatusError** – When creating the draft without basing on a branch.

get_draft(*draft_number: Optional[int] = None*) → `tensorbay.client.struct.Draft`

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** – 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.

list_drafts() → *tensorbay.client.lazy.PagingList[tensorbay.client.struct.Draft]*

List all the drafts.

Returns The *PagingList* of *drafts*.

update_draft(*draft_number: Optional[int] = None, *, title: Optional[str] = None, description: Optional[str] = None*) → None

Update the draft.

Parameters

- **draft_number** – The updated draft number. If is not given, update the current draft.
- **title** – The title of the draft.
- **description** – The description of the draft.

close_draft(*draft_number: Optional[int] = None*) → None

Close the draft.

Parameters **draft_number** – The closed draft number. If is not given, close the current draft.

get_commit(*revision: Optional[str] = None*) → *tensorbay.client.struct.Commit*

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** – 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.

list_commits(*revision: Optional[str] = None*) → *tensorbay.client.lazy.PagingList[tensorbay.client.struct.Commit]*

List the commits.

Parameters **revision** – 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.

Returns The *PagingList* of *commits*.

create_branch(*name: str, revision: Optional[str] = None*) → 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** – The branch name.

- **revision** – 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.

get_branch(name: str) → *tensorbay.client.struct.Branch*

Get the branch with the given name.

Parameters **name** – 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.

list_branches() → *tensorbay.client.lazy.PagingList[tensorbay.client.struct.Branch]*

List the information of branches.

Returns The PagingList of *branches*.

delete_branch(name: str) → None

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** – The name of the branch to be deleted.

Raises **OperationError** – When deleting the current branch.

create_tag(name: str, revision: Optional[str] = None) → 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** – The tag name to be created for the specific commit.
- **revision** – 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.

get_tag(name: str) → *tensorbay.client.struct.Tag*

Get the certain tag with the given name.

Parameters **name** – 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.

list_tags() → *tensorbay.client.lazy.PagingList[tensorbay.client.struct.Tag]*

List the information of tags.

Returns The PagingList of *tags*.

delete_tag(name: str) → None

Delete a tag.

Parameters **name** – The tag name to be deleted for the specific commit.

1.20.2 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: Optional[float] = None)
```

Bases: *tensorbay.utility.repr.ReprMixin*

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

Parameters **timestamp** – The timestamp for the file.

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: str, *, target_remote_path: Optional[str] = None,
                                  timestamp: Optional[float] = 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** – The file local path.
- **target_remote_path** – The file remote path after uploading to tensorbay.
- **timestamp** – The timestamp for the file.

path

The file local path.

timestamp

The timestamp for the file.

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() → Dict[str, Any]

Get the callback request body for uploading.

Returns

The callback request body, which look like:

(continued from previous page)

```

        "POLYGON": {...},
        "POLYLINE2D": {...},
        "KEYPOINTS2D": {...},
        "SENTENCE": {...}
    }
}

```

- **_url_getter** – The url getter of the remote file.

Returns The loaded *RemoteData* object.

```
class tensorbay.dataset.data.AuthData(
    cloud_path: str, *, target_remote_path: Optional[str] = None,
    timestamp: Optional[float] = None, _url_getter:
    Optional[Callable[[str], str]] = None)

```

Bases: *tensorbay.dataset.data.DataBase*, *tensorbay.utility.file.RemoteFileMixin*

AuthData is a combination of a specific cloud stored 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** – The cloud file path.
- **target_remote_path** – The file remote path after uploading to tensorbay.
- **timestamp** – The timestamp for the file.
- **_url_getter** – The url getter of the remote file.

path

The cloud file path.

timestamp

The timestamp for the file.

label

The *Label* instance that contains all the label information of the file.

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: bool = False, bin_point_cloud_fields:
    Optional[Iterable[str]] = None)

```

Bases: *tensorbay.utility.attr.AttrsMixin*, *tensorbay.utility.repr.ReprMixin*

This is a class stores the basic information of *DatasetBase*.

Parameters

- **is_continuous** – Whether the data inside the dataset is time-continuous.
- **bin_point_cloud_fields** – The field names of the bin point cloud files in the dataset.

classmethod loads(*contents: Dict[str, Any]*) → `tensorbay.dataset.dataset._T`

Loads a [Notes](#) instance from the given contents.

Parameters contents – The given dict containing the dataset notes:

```
{
    "isContinuous":          <boolean>
    "binPointCloudFields": [ <array> or null
                           <field_name>, <str>
                           ...
    ]
}
```

Returns The loaded [Notes](#) instance.

keys() → `KeysView[str]`

Return the valid keys within the notes.

Returns The valid keys within the notes.

dumps() → `Dict[str, Any]`

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>
                           ...
    ]
}
```

class `tensorbay.dataset.dataset.DatasetBase`(*name: str, gas: Optional[GAS] = None, revision: Optional[str] = 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.

keys() → Tuple[str, ...]

Get all segment names.

Returns A tuple containing all segment names.

load_catalog(*filepath: str*) → None

Load catalog from a json file.

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

add_segment(*segment: tensorbay.dataset.dataset._T*) → None

Add a segment to the dataset.

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

class tensorbay.dataset.dataset.**Dataset**(*name: str, gas: Optional[GAS] = None, revision: Optional[str] = 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: str = 'default'*) → *tensorbay.dataset.segment.Segment*

Create a segment with the given name.

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

Returns The created *Segment*.

class tensorbay.dataset.dataset.**FusionDataset**(*name: str, gas: Optional[GAS] = None, revision: Optional[str] = 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: str = 'default'*) → *tensorbay.dataset.segment.FusionSegment*

Create a fusion segment with the given name.

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

Returns The created *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: str = 'default', client: Optional[DatasetClient] = None*)
Bases: *tensorbay.utility.name.NameMixin, tensorbay.utility.user.UserMutableSequence[DataBase._Type]*

This class defines the concept of segment.

Segment is a concept in [Dataset](#). It is the structure that composes [Dataset](#), and consists of a series of [Data](#) without sensor information.

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

Since [Segment](#) extends [UserMutableSequence](#), its basic operations are the same as a list's.

To initialize a Segment and add a [Data](#) to it:

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

Parameters

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

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

Sort the list in ascending order and return None.

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

Parameters

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

Raises `NotImplementedError` – The sort method for segment init from client is not supported yet.

```
class tensorbay.dataset.segment.FusionSegment(name: str = 'default', client:
                                             Optional[FusionDatasetClient] = 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](#) corresponding 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: Optional[ulid.ulid.ULID] = 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: Dict[str, Any], frame_index: int, urls: Sequence[Mapping[str, str]]) → tensorbay.dataset.frame._T`

Loads a *Frame* object from a response body.

Parameters

- **body** – The response body which contains the information of a frame, whose format should be like:

```
{
  "frameId": <str>,
  "frame": [
    {
      "sensorName": <str>,
      "remotePath": <str>,
      "timestamp": <float>,
      "label": {...}
    },
    ...
  ]
}
```

- **frame_index** – The index of the frame.
- **urls** – A sequence of mappings which key is the sensor name and value is the url.

Returns The loaded *Frame* object.

1.20.3 tensorbay.geometry

tensorbay.geometry.box

Box2D, Box3D.

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

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

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

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

This class defines the concept of Box2D.

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

Parameters

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

Examples

```
>>> Box2D(1, 2, 3, 4)
Box2D(1, 2, 3, 4)
```

static *iou(box1: tensorbay.geometry.box.Box2D, box2: tensorbay.geometry.box.Box2D)* → float

Calculate the intersection over union of two 2D boxes.

Parameters

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

Returns The intersection over union between the two input boxes.

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: float, y: float, width: float, height: float) → tensorbay.geometry.box._B2`
Create a [Box2D](#) instance from the top-left vertex and the width and the height.

Parameters

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

Returns The created [Box2D](#) instance.

Examples

```
>>> Box2D.from_xywh(1, 2, 3, 4)
Box2D(1, 2, 4, 6)
```

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

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

Returns The loaded [Box2D](#) object.

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() → Dict[str, float]

Dumps a 2D box into a dict.

Returns A dict containing vertex coordinates of the box.

Examples

```
>>> box2d = Box2D(1, 2, 3, 4)
>>> box2d.dumps()
{'xmin': 1, 'ymin': 2, 'xmax': 3, 'ymax': 4}
```

area() → float

Return the area of the 2D box.

Returns The area of the 2D box.

Examples

```
>>> box2d = Box2D(1, 2, 3, 4)
>>> box2d.area()
4
```

```
class tensorbay.geometry.box.Box3D(size: Iterable[float], translation: Iterable[float] = (0, 0, 0), rotation:
    Union[Iterable[float], quaternion.quaternion] = (1, 0, 0, 0), *,
    transform_matrix: Optional[Union[Sequence[Sequence[float]],
    numpy.ndarray]] = 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** – Translation in a sequence of [x, y, z].
- **rotation** – Rotation in a sequence of [w, x, y, z] or numpy quaternion.
- **size** – Size in a sequence of [x, y, z].
- **transform_matrix** – A 4x4 or 3x4 transform matrix.

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: Dict[str, Dict[str, float]]) → tensorbay.geometry.box._B3

Load a [Box3D](#) from a dict containing the coordinates of the 3D box.

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

Returns The loaded [Box3D](#) object.

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: tensorbay.geometry.box.Box3D, box2: tensorbay.geometry.box.Box3D, angle_threshold: float = 5) → float`

Calculate the intersection over union between two 3D boxes.

Parameters

- **box1** – A 3D box.
- **box2** – A 3D box.
- **angle_threshold** – The threshold of the relative angles between two input 3d boxes in degree.

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

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() → float

Return the volume of the 3D box.

Returns The volume of the 3D box.

Examples

```
>>> box3d = Box3D(size=(1, 2, 3))
>>> box3d.volume()
6
```

dumps() → Dict[str, Dict[str, float]]

Dumps the 3D box into a dict.

Returns A dict containing translation, rotation and size information.

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: float, **kwargs: float)
Bases: *tensorbay.utility.user.UserSequence*[float]

This class defines the concept of Keypoint2D.

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

Parameters

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

Visible status can be “BINARY” or “TERNARY”:

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

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: Dict[str, float]) → tensorbay.geometry.keypoint._T

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

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

Returns The loaded *Keypoint2D* object.

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() → Dict[str, float]

Dumps the *Keypoint2D* into a dict.

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

Examples

```
>>> keypoint = Keypoint2D(1.0, 2.0, 1)
>>> keypoint.dumps()
{'x': 1.0, 'y': 2.0, 'v': 1}
```

class tensorbay.geometry.keypoint.**Keypoints2D**(points: Optional[Iterable[Iterable[float]]] = 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: List[Dict[str, float]]) → *tensorbay.geometry.keypoint._P*

Load a *Keypoints2D* from a list of dict.

Parameters contents – A list of dictionaries containing 2D keypoint.

Returns The loaded *Keypoints2D* object.

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: Optional[Iterable[Iterable[float]]] = 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: List[Dict[str, float]]) → tensorbay.geometry.point_list._P

Load a *PointList2D* from a list of dictionaries.

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

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

Returns The loaded *PointList2D* object.

dumps() → List[Dict[str, float]]

Dumps a *PointList2D* into a point list.

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

bounds() → [tensorbay.geometry.box.Box2D](#)

Calculate the bounds of point list.

Returns The bounds of point list.

class tensorbay.geometry.point_list.**MultiPointList2D**(point_lists: Optional[Iterable[Iterable[Iterable[float]]]] = 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*: List[List[Dict[str, float]]]) → *tensorbay.geometry.point_list._P*

Loads a *MultiPointList2D* from the given contents.

Parameters *contents* – A list of dictionary lists containing the coordinates of the vertexes of the multiple point lists:

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

Returns The loaded *MultiPointList2D* object.

dumps() → List[List[Dict[str, float]]]

Dumps all the information of the *MultiPointList2D*.

Returns All the information of the *MultiPointList2D*.

bounds() → *tensorbay.geometry.box.Box2D*

Calculate the bounds of multiple point lists.

Returns The bounds of multiple point lists.

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*: Optional[Iterable[Iterable[float]]] = 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: List[Dict[str, float]])` → `tensorbay.geometry.polygon._P`

Loads the information of *Polygon*.

Parameters `contents` – A list of dictionary lists containing the coordinates of the vertexes of the polygon.

Returns The loaded *Polygon* object.

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() → `float`

Return the area of the polygon.

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

Returns The area of the polygon.

Examples

```
>>> polygon = Polygon([[1, 2], [2, 2], [2, 3]])
>>> polygon.area()
0.5
```

class `tensorbay.geometry.polygon.MultiPolygon(polygons: Optional[Iterable[Iterable[Iterable[float]]]])`

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: List[List[Dict[str, float]]])` → `tensorbay.geometry.polygon._P`

Loads a *MultiPolygon* from the given contents.

Parameters **contents** – A list of dict lists containing the coordinates of the vertices of the polygon list.

Returns The loaded MultiPolyline2D object.

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() → List[List[Dict[str, float]]]

Dumps a *MultiPolygon* into a polygon list.

Returns All the information of the *MultiPolygon*.

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: Optional[Iterable[int]])

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: List[int]) → *tensorbay.geometry.polygon.RLE*

Loads a :class:`RLE` from the given contents.

Parameters `contents` – One rle mask.

Returns The loaded *RLE* object.

Examples

```
>>> contents = [272, 2, 4, 4, 2, 9]
>>> rle = RLE.loads(contents)
>>> rle
RLE [
  272,
  2,
  ...
]
```

dumps() → List[int]

Dumps a *RLE* into one rle mask.

Returns All the information of the *RLE*.

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: Optional[Iterable[Iterable[float]]] = 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: Sequence[Sequence[float]], polyline2:
Sequence[Sequence[float]]) → float

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

Parameters

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

Returns The computed distance between the two polylines.

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: Sequence[Sequence[float]], polyline2: Sequence[Sequence[float]]*) → float
Calculate the similarity between two polylines, range from 0 to 1.

Parameters

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

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

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: List[Dict[str, float]]*) → tensorbay.geometry.polyline._P
Load a [Polyline2D](#) from a list of dict.

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

Returns The loaded [Polyline2D](#) object.

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: Optional[Iterable[Iterable[Iterable[float]]]] = 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: List[List[Dict[str, float]]])` → `tensorbay.geometry.polyline._P`
Loads a *MultiPolyline2D* from the given contents.

Parameters `contents` – A list of dict lists containing the coordinates of the vertexes of the polyline list.

Returns The loaded *MultiPolyline2D* object.

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() → `List[List[Dict[str, float]]]`
Dumps a *MultiPolyline2D* into a polyline list.

Returns All the information of the *MultiPolyline2D*.

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: Iterable[float] = (0, 0, 0), rotation:
                                             Union[Iterable[float], quaternion.quaternion] = (1, 0,
                                             0, 0), *, matrix:
                                             Optional[Union[Sequence[Sequence[float]],
                                             numpy.ndarray]] = None)
```

Bases: *tensorbay.utility.repr.ReprMixin*

This class defines the concept of Transform3D.

Transform3D contains rotation and translation of the 3D transform.

Parameters

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

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

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: Dict[str, Dict[str, float]]*) → *tensorbay.geometry.transform._T*

Load a *Transform3D* from a dict containing rotation and translation.

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

Returns The loaded *Transform3D* object.

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() → Dict[str, Dict[str, float]]

Dumps the *Transform3D* into a dict.

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

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: float, y: float, z: float) → None

Set the translation of the transform.

Parameters

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

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: *Optional[float] = None*, x: *Optional[float] = None*, y: *Optional[float] = None*, z: *Optional[float] = None*, *, quaternion: *Optional[quaternion.quaternion] = None*) → None

Set the rotation of the transform.

Parameters

- **w** – The w componet of the roation quaternion.
- **x** – The x componet of the roation quaternion.
- **y** – The y componet of the roation quaternion.
- **z** – The z componet of the roation quaternion.
- **quaternion** – Numpy quaternion representing the rotation.

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() → numpy.ndarray

Return the transform as a 4x4 transform matrix.

Returns A 4x4 numpy array represents the transform matrix.

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() → `tensorbay.geometry.transform._T`
Return the inverse of the transform.

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

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: float, y: float, z: Optional[float] = None*)
Bases: `tensorbay.utility.user.UserSequence`[float]

This class defines the basic concept of Vector.

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

Parameters

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

Examples

```
>>> Vector(1, 2)
Vector2D(1, 2)
```

```
>>> Vector(1, 2, 3)
Vector3D(1, 2, 3)
```

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

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

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

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

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: float, **kwargs: float)
Bases: *tensorbay.utility.user.UserSequence*[float]

This class defines the concept of Vector2D.

Vector2D contains the coordinates of a 2D vector.

Parameters

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

Examples

```
>>> Vector2D(1, 2)
Vector2D(1, 2)
```

classmethod loads(*contents: Dict[str, float]*) → *tensorbay.geometry.vector._V2*

Load a *Vector2D* object from a dict containing coordinates of a 2D vector.

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

Returns The loaded *Vector2D* object.

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() → Dict[str, float]

Dumps the vector into a dict.

Returns A dict containing the vector coordinate.

Examples

```
>>> vector_2d = Vector2D(1, 2)
>>> vector_2d.dumps()
{'x': 1, 'y': 2}
```

class tensorbay.geometry.vector.**Vector3D**(*args: float, **kwargs: float)

Bases: [tensorbay.utility.user.UserSequence](#)[float]

This class defines the concept of Vector3D.

[Vector3D](#) contains the coordinates of a 3D Vector.

Parameters

- **x** – The x coordinate of the 3D vector.
- **y** – The y coordinate of the 3D vector.
- **z** – The z coordinate of the 3D vector.

Examples

```
>>> Vector3D(1, 2, 3)
Vector3D(1, 2, 3)
```

classmethod `loads(contents: Dict[str, float])` → `tensorbay.geometry.vector._V3`

Load a `Vector3D` object from a dict containing coordinates of a 3D vector.

Parameters `contents` – A dict contains coordinates of a 3D vector.

Returns The loaded `Vector3D` object.

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() → Dict[str, float]

Dumps the vector into a dict.

Returns A dict containing the vector coordinates.

Examples

```
>>> vector_3d = Vector3D(1, 2, 3)
>>> vector_3d.dumps()
{'x': 1, 'y': 2, 'z': 3}
```

1.20.4 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_: Union[str, None, Type[Optional[Union[list, bool, int, float, str]]], Iterable[Union[str, None, Type[Optional[Union[list, bool, int, float, str]]]]] = "", enum: Optional[Iterable[Optional[Union[str, float, bool]]]] = None, minimum: Optional[float] = None, maximum: Optional[float] = None, items: Optional[tensorbay.label.attributes.Items] = None)
```

Bases: *tensorbay.utility.repr.ReprMixin*, *tensorbay.utility.common.EqMixin*

The base class of *AttributeInfo*, representing the items of an attribute.

When the value type of an attribute is array, the *AttributeInfo* would contain an 'items' field.

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** – All the possible values of an enumeration attribute.
- **minimum** – The minimum value of number type attribute.
- **maximum** – The maximum value of number type attribute.
- **items** – The items inside array type attributes.

type

The type of the attribute value, could be a single type or multi-types.

enum

All the possible values of an enumeration attribute.

minimum

The minimum value of number type attribute.

maximum

The maximum value of number type attribute.

items

The items inside array type attributes.

Raises `TypeError` – When both `enum` and `type_` are absent or when `type_` is array and `items` is absent.

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: Dict[str, Any])` → `tensorbay.label.attributes._T`

Load an `Items` from a dict containing the items information.

Parameters **contents** – A dict containing the information of the items.

Returns The loaded `Items` object.

Examples

```
>>> contents = {
...     "type": "array",
...     "enum": [1, 2, 3, 4, 5],
...     "minimum": 1,
...     "maximum": 5,
...     "items": {
...         "enum": [None],
...         "type": "null",
...     }
... }
```

(continues on next page)

(continued from previous page)

```

...     },
... }
>>> Items.loads(contents)
Items(
  (type): 'array',
  (enum): [...],
  (minimum): 1,
  (maximum): 5,
  (items): Items(...)
)

```

dumps() → Dict[str, Any]

Dumps the information of the items into a dict.

Returns A dict containing all the information of the items.

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: str, *, type_: Union[str, None,
    Type[Optional[Union[list, bool, int, float, str]]],
    Iterable[Union[str, None, Type[Optional[Union[list,
    bool, int, float, str]]]]] = "", enum:
    Optional[Iterable[Optional[Union[str, float, bool]]]] =
    None, minimum: Optional[float] = None, maximum:
    Optional[float] = None, items:
    Optional[tensorbay.label.attributes.Items] = None,
    parent_categories: Union[None, str, Iterable[str]] =
    None, description: str = "")

```

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** – The name of the attribute.
- **type** – The type of the attribute value, could be a single type or multi-types. The type must be within the followings:
 - array
 - boolean
 - integer
 - number

- string
- null
- instance
- **enum** – All the possible values of an enumeration attribute.
- **minimum** – The minimum value of number type attribute.
- **maximum** – The maximum value of number type attribute.
- **items** – The items inside array type attributes.
- **parent_categories** – The parent categories of the attribute.
- **description** – The description of the attribute.

type

The type of the attribute value, could be a single type or multi-types.

enum

All the possible values of an enumeration attribute.

minimum

The minimum value of number type attribute.

maximum

The maximum value of number type attribute.

items

The items inside array type attributes.

parent_categories

The parent categories of the attribute.

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): [
```

(continues on next page)

(continued from previous page)

```

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: Dict[str, Any])` → `tensorbay.label.attributes._T`

Load an `AttributeInfo` from a dict containing the attribute information.

Parameters `contents` – A dict containing the information of the attribute.

Returns The loaded `AttributeInfo` object.

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'
    ]
)

```

method `dumps()` → `Dict[str, Any]`

Dumps the information of this attribute into a dict.

Returns A dict containing all the information of this attribute.

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: str = ""*)

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** – The description of the entire subcatalog.

description

The description of the entire subcatalog.

Type str

classmethod **loads**(*contents: Dict[str, Any]*) → tensorbay.label.basic._T

Loads a subcatalog from a dict containing the information of the subcatalog.

Parameters **contents** – A dict containing the information of the subcatalog.

Returns The loaded [SubcatalogBase](#) object.

dumps() → Dict[str, Any]

Dumps all the information of the subcatalog into a dict.

Returns A dict containing all the information of the subcatalog.

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	explanation
<i>ClassificationSubcatalog</i>	subcatalog for classification type of label
<i>Box2DSubcatalog</i>	subcatalog for 2D bounding box type of label
<i>Box3DSubcatalog</i>	subcatalog for 3D bounding box type of label
<i>Keypoints2DSubcatalog</i>	subcatalog for 2D keypoints type of label
<i>PolygonSubcatalog</i>	subcatalog for polygon type of label
<i>Polyline2DSubcatalog</i>	subcatalog for 2D polyline type of label
<i>MultiPolygonSubcatalog</i>	subcatalog for multiple polygon type of label
<i>RLESubcatalog</i>	subcatalog for rle mask type of label
<i>MultiPolyline2DSubcatalog</i>	subcatalog for 2D multiple polyline type of label
<i>SentenceSubcatalog</i>	subcatalog for transcribed 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: Dict[str, Any]*) → *tensorbay.label.catalog._T*

Load a Catalog from a dict containing the catalog information.

Parameters **contents** – A dict containing all the information of the catalog.

Returns The loaded *Catalog* object.

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() → Dict[str, Any]

Dumps the catalog into a dict containing the information of all the subcatalog.

Returns A dict containing all the subcatalog information with their label types as keys.

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.

Table 1.9: label classes

label classes	explanation
<i>Classification</i>	classification type of label
<i>LabeledBox2D</i>	2D bounding box type of label
<i>LabeledBox3D</i>	3D bounding box type of label
<i>LabeledPolygon</i>	polygon type of label
<i>LabeledMultiPolygon</i>	polygon lists type of label
<i>LabeledRLE</i>	rle mask type of label
<i>LabeledPolyline2D</i>	2D polyline type of label
<i>LabeledMultiPolyline2D</i>	2D polyline lists type of label
<i>LabeledKeypoints2D</i>	2D keypoints type of label
<i>LabeledSentence</i>	transcribed 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: Dict[str, Any]*) → tensorbay.label.label._T

Loads data from a dict containing the labels information.

Parameters *contents* – A dict containing the labels information.

Returns A *Label* instance containing labels information from the given dict.

Examples

```
>>> contents = {
...     "CLASSIFICATION": {
...         "category": "example_category",
...         "attributes": {"example_attribute1": "a"}
...     }
... }
>>> Label.loads(contents)
Label(
  (classification): Classification(
    (category): 'example_category',
    (attributes): {...}
  )
)
```

dumps() → Dict[str, Any]

Dumps all labels into a dict.

Returns Dumped labels dict.

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: bool = 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** – A boolean value indicates whether the corresponding subcatalog contains tracking information.

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
```

(continues on next page)

(continued from previous page)

```
Box2DSubcatalog(
    (is_tracking): True,
    (category_delimiter): '.',
    (categories): NameList [...],
    (attributes): NameList [...]
)
```

```
class tensorbay.label.label_box.LabeledBox2D(xmin: float, ymin: float, xmax: float, ymax: float, *,
                                              category: Optional[str] = None, attributes:
                                              Optional[Dict[str, Any]] = None, instance: Optional[str]
                                              = None)
```

Bases: [tensorbay.utility.user.UserSequence](#)[float]

This class defines the concept of 2D bounding box label.

[LabeledBox2D](#) is the 2D bounding box type of label, which is often used for CV tasks such as object detection.

Parameters

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

```
>>> xmin, ymin, xmax, ymax = 1, 2, 4, 4
>>> LabeledBox2D(
...     xmin,
...     ymin,
...     xmax,
...     ymax,
...     category="example",
...     attributes={"attr": "a"},
...     instance="12345",
```

(continues on next page)

(continued from previous page)

```

... )
LabeledBox2D(1, 2, 4, 4)(
    (category): 'example',
    (attributes): {...},
    (instance): '12345'
)

```

classmethod `from_xywh`(*x: float, y: float, width: float, height: float, *, category: Optional[str] = None, attributes: Optional[Dict[str, Any]] = None, instance: Optional[str] = None*) → `tensorbay.label.label_box._T`

Create a `LabeledBox2D` instance from the top-left vertex, the width and height.

Parameters

- **x** – X coordinate of the top left vertex of the box.
- **y** – Y coordinate of the top left vertex of the box.
- **width** – Length of the box along the x axis.
- **height** – Length of the box along the y axis.
- **category** – The category of the label.
- **attributes** – The attributes of the label.
- **instance** – The instance id of the label.

Returns The created `LabeledBox2D` instance.

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: Dict[str, Any]*) → `tensorbay.label.label_box._T`

Loads a `LabeledBox2D` from a dict containing the information of the label.

Parameters **contents** – A dict containing the information of the 2D bounding box label.

Returns The loaded `LabeledBox2D` object.

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',
  (attributes): {...},
  (instance): '12345'
)
```

dumps() → Dict[str, Any]

Dumps the current 2D bounding box label into a dict.

Returns A dict containing all the information of the 2D box label.

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: bool = 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** – A boolean value indicates whether the corresponding subcatalog contains tracking information.

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
```

(continues on next page)

(continued from previous page)

```
Box3DSubcatalog(
    (is_tracking): True,
    (category_delimiter): '.',
    (categories): NameList [...],
    (attributes): NameList [...]
)
```

```
class tensorbay.label.label_box.LabeledBox3D(size: Iterable[float], translation: Iterable[float] = (0, 0, 0), rotation: Union[Iterable[float], quaternion.quaternion] = (1, 0, 0, 0), *, transform_matrix: Optional[Union[Sequence[Sequence[float]], numpy.ndarray]] = None, category: Optional[str] = None, attributes: Optional[Dict[str, Any]] = None, instance: Optional[str] = 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** – Size of the 3D bounding box label in a sequence of [x, y, z].
- **translation** – Translation of the 3D bounding box label in a sequence of [x, y, z].
- **rotation** – Rotation of the 3D bounding box label in a sequence of [w, x, y, z] or a numpy quaternion object.
- **transform_matrix** – A 4x4 or 3x4 transformation matrix.
- **category** – Category of the 3D bounding box label.
- **attributes** – Attributes of the 3D bounding box label.
- **instance** – The instance id of the 3D bounding box label.

category

The category of the label.

Type str

attributes

The attributes of the label.

Type Dict[str, 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(
  (size): Vector3D(1, 2, 3),
  (translation): Vector3D(1, 2, 3),
  (rotation): quaternion(0, 1, 0, 0),
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

classmethod `loads(contents: Dict[str, Any]) → tensorbay.label.label_box._T`
Loads a `LabeledBox3D` from a dict containing the information of the label.

Parameters `contents` – A dict containing the information of the 3D bounding box label.

Returns The loaded `LabeledBox3D` object.

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'
)
```

dump`s()` → Dict[str, Any]
Dumps the current 3D bounding box label into a dict.

Returns A dict containing all the information of the 3D bounding box label.

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: str = ""*)
 Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.CategoriesMixin*,
tensorbay.label.supports.AttributesMixin

This class defines the subcatalog for classification type of labels.

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: Optional[str] = None,
                                                         attributes: Optional[Dict[str, Any]] =
                                                         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** – The category of the label.
- **attributes** – The attributes of the label.

category

The category of the label.

Type str

attributes

The attributes of the label.

Type Dict[str, 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: Dict[str, Any]*) → tensorbay.label.label_classification._T

Loads a Classification label from a dict containing the label information.

Parameters **contents** – A dict containing the information of the classification label.

Returns The loaded *Classification* object.

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: bool = 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** – A boolean value indicates whether the corresponding subcatalog contains tracking information.

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"]))
```

(continues on next page)

(continued from previous page)

```

>>> 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: int, *, names: Optional[Iterable[str]] = None, skeleton: Optional[Iterable[Iterable[int]]] = None, visible: Optional[str] = None, parent_categories: Union[None, str, Iterable[str]] = None, description: str = ""*) → None

Add a type of keypoints to the subcatalog.

Parameters

- **number** – The number of keypoints.
- **names** – All the names of keypoints.
- **skeleton** – The skeleton of the keypoints indicating which keypoint should connect with another.
- **visible** – The visible type of the keypoints, can only be ‘BINARY’ or ‘TERNARY’. It determines the range of the *Keypoint2D.v*.
- **parent_categories** – The parent categories of the keypoints.
- **description** – The description of keypoints.

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() → Dict[str, Any]

Dumps all the information of the keypoints into a dict.

Returns A dict containing all the information of this Keypoints2DSubcatalog.

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:
    Optional[Iterable[Iterable[float]]] =
    None, *, category: Optional[str] = None,
    attributes: Optional[Dict[str, Any]] =
    None, instance: Optional[str] = None)
```

Bases: `tensorbay.geometry.point_list.PointList2D[tensorbay.geometry.keypoint.Keypoint2D]`

This class defines the concept of 2D keypoints label.

`LabeledKeypoints2D` is the 2D keypoints type of label, which is often used for CV tasks such as human body pose estimation.

Parameters

- **keypoints** – A list of 2D keypoint.
- **category** – The category of the label.
- **attributes** – The attributes of the label.
- **instance** – The instance id of the label.

category

The category of the label.

Type str

attributes

The attributes of the label.

Type Dict[str, 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: Dict[str, Any]*) → tensorbay.label.label_keypoints._T

Loads a LabeledKeypoints2D from a dict containing the information of the label.

Parameters contents – A dict containing the information of the 2D keypoints label.

Returns The loaded *LabeledKeypoints2D* object.

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() → Dict[str, Any]

Dumps the current 2D keypoints label into a dict.

Returns A dict containing all the information of the 2D keypoints label.

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: str = ""*)

Bases: [tensorbay.label.basic.SubcatalogBase](#), [tensorbay.label.supports.MaskCategoriesMixin](#), [tensorbay.label.supports.AttributesMixin](#)

This class defines the subcatalog for semantic mask type of labels.

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: str = ""*)

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.

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'}],
...     }
... }
>>> 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: str = "")
    Bases: tensorbay.label.basic.SubcatalogBase, tensorbay.label.supports.MaskCategoriesMixin, tensorbay.label.supports.AttributesMixin
```

This class defines the subcatalog for panoptic mask type of labels.

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(
```

(continues on next page)

(continued from previous page)

```
(categories): NameList [...],
(attributes): NameList [...]
)
```

class tensorbay.label.label_mask.SemanticMaskBaseBases: [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.InstanceMaskBaseBases: [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.PanopticMaskBaseBases: [tensorbay.utility.repr.ReprMixin](#)

PanopticMaskBase is a base class for the panoptic 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]]]]]**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: str)Bases: [tensorbay.label.label_mask.SemanticMaskBase](#), [tensorbay.utility.file.FileMixin](#)

SemanticMask is a class for the local 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]]]]]**get_callback_body()** → Dict[str, Any]

Get the callback request body for uploading.

Returns

The callback request body, which looks like:

```
{
  "checksum": <str>,
  "fileSize": <int>,
```

(continues on next page)

(continued from previous page)

```

    "info": [
        {
            "categoryId": 0,
            "attributes": {
                "occluded": True
            }
        },
        {
            "categoryId": 1,
            "attributes": {
                "occluded": False
            }
        }
    ]
}

```

class tensorbay.label.label_mask.**InstanceMask**(*local_path: str*)

Bases: [tensorbay.label.label_mask.InstanceMaskBase](#), [tensorbay.utility.file.FileMixin](#)

InstanceMask is a class for the local 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]]]]]

get_callback_body() → Dict[str, Any]

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
            }
        }
    ]
}

```

class tensorbay.label.label_mask.**PanopticMask**(*local_path: str*)

Bases: [tensorbay.label.label_mask.PanopticMaskBase](#), [tensorbay.utility.file.FileMixin](#)

PanopticMask is a class for the local panoptic 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]]]]]

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() → Dict[str, Any]

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
      }
    }
  ]
}
```

class tensorbay.label.label_mask.RemoteSemanticMask(remote_path: str, *, _url_getter: Optional[Callable[[str], str]] = None)

Bases: [tensorbay.label.label_mask.SemanticMaskBase](#), [tensorbay.utility.file.RemoteFileMixin](#)

RemoteSemanticMask is a class for the remote 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]]]]]

classmethod from_response_body(body: Dict[str, Any]) → tensorbay.label.label_mask._T

Loads a [RemoteSemanticMask](#) object from a response body.

Parameters body – 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
      }
    },
    {
      "categoryId": 1,
      "attributes": {
        "occluded": False
      }
    }
  ]
}
```

Returns The loaded *RemoteSemanticMask* object.

class tensorbay.label.label_mask.**RemoteInstanceMask**(remote_path: str, *, _url_getter: Optional[Callable[[str], str]] = None)

Bases: *tensorbay.label.label_mask.InstanceMaskBase*, *tensorbay.utility.file.RemoteFileMixin*

RemoteInstanceMask is a class for the remote 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]]]]]

classmethod from_response_body(body: Dict[str, Any]) → tensorbay.label.label_mask._T

Loads a *RemoteInstanceMask* object from a response body.

Parameters **body** – 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
      }
    }
  ]
}
```

Returns The loaded *RemoteInstanceMask* object.

```
class tensorbay.label.label_mask.RemotePanopticMask(remote_path: str, *, _url_getter:
                                                    Optional[Callable[[str], str]] = None)
    Bases: tensorbay.label.label_mask.PanopticMaskBase, tensorbay.utility.file.
            RemoteFileMixin
```

RemotePanopticMask is a class for the remote panoptic 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]]]]]

```
classmethod from_response_body(body: Dict[str, Any]) → tensorbay.label.label_mask._T
```

Loads a *RemotePanopticMask* object from a response body.

Parameters *body* – 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.

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: bool = 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* – A boolean value indicates whether the corresponding subcatalog contains tracking information.

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 [...],
  (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
```

(continues on next page)

(continued from previous page)

```
>>> polygon_subcatalog
PolygonSubcatalog(
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

class `tensorbay.label.label_polygon.MultiPolygonSubcatalog`(*is_tracking: bool = 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** – A boolean value indicates whether the corresponding subcatalog contains tracking information.

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,
```

(continues on next page)

(continued from previous page)

```
(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: bool = 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** – A boolean value indicating whether the corresponding subcatalog contains tracking information.

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: Optional[Iterable[Iterable[float]]] =
                                                    None, *, category: Optional[str] = None,
                                                    attributes: Optional[Dict[str, Any]] = None,
                                                    instance: Optional[str] = None)
```

Bases: `tensorbay.geometry.point_list.PointList2D[tensorbay.geometry.vector.Vector2D]`

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 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

```
>>> 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: Dict[str, Any]*) → tensorbay.label.label_polygon._T

Loads a LabeledPolygon from a dict containing the information of the label.

Parameters **contents** – A dict containing the information of the polygon label.

Returns The loaded *LabeledPolygon* object.

Examples

```
>>> contents = {
...     "polygon": [
...         {"x": 1, "y": 2},
...         {"x": 2, "y": 3},
...         {"x": 1, "y": 3},
...     ],
...     "category": "example",
...     "attributes": {"key": "value"},
...     "instance": "12345",
... }
>>> LabeledPolygon.loads(contents)
LabeledPolygon [
  Vector2D(1, 2),
  Vector2D(2, 3),
  Vector2D(1, 3)
](
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

dumps() → Dict[str, Any]

Dumps the current polygon label into a dict.

Returns A dict containing all the information of the polygon label.

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: Op-  
                                                         tional[Iterable[Iterable[Iterable[float]]]] =  
                                                         None, *, category: Optional[str] = None,  
                                                         attributes: Optional[Dict[str, Any]] =  
                                                         None, instance: Optional[str] = None)
```

Bases: [tensorbay.geometry.point_list.MultiPointList2D](#)[\[tensorbay.geometry.polygon.Polygon\]](#)

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: Dict[str, Any]) → tensorbay.label.label_polygon._T`

Loads a LabeledMultiPolygon from a list of dict containing the information of the label.

Parameters `contents` – A dict containing the information of the multipolygon label.

Returns The loaded *LabeledMultiPolygon* object.

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",
... }
>>> LabeledMultiPolygon.loads(contents)
LabeledMultiPolygon [
  Polygon [...],
  Polygon [...],
](
  (category): 'example',
  (attributes): {...},
  (instance): '12345'
)
```

dumps() → Dict[str, Any]

Dumps the current multipolygon label into a dict.

Returns A dict containing all the information of the multipolygon label.

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: Optional[Iterable[int]] = None*, *, *category: Optional[str] = None*, *attributes: Optional[Dict[str, Any]] = None*, *instance: Optional[str] = None*)

Bases: `tensorbay.utility.user.UserMutableSequence[int]`

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 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

```
>>> 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: Dict[str, Any])` → `tensorbay.label.label_polygon._T`

Loads a `LabeledRLE` from a dict containing the information of the label.

Parameters `contents` – A dict containing the information of the rle label.

Returns The loaded `LabeledRLE` object.

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): {...},
  (instance): '12345'
)
```

dumps() → `Dict[str, Any]`

Dumps the current rle label into a dict.

Returns A dict containing all the information of the rle label.

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: bool = 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** – A boolean value indicates whether the corresponding subcatalog contains tracking information.

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

Examples

Initialization Method 1: Init from Polyline2DSubcatalog.loads() method.

```
>>> catalog = {
...     "POLYLINE2D": {
...         "isTracking": True,
...         "categories": [{"name": "0"}, {"name": "1"}],
...         "attributes": [{"name": "gender", "enum": ["male", "female"]}],
...     }
... }
>>> Polyline2DSubcatalog.loads(catalog["POLYLINE2D"])
Polyline2DSubcatalog(
  (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
>>> polyline2d_subcatalog.categories = categories
>>> polyline2d_subcatalog.attributes = attributes
>>> polyline2d_subcatalog
Polyline2DSubcatalog(
  (is_tracking): True,
  (categories): NameList [...],
  (attributes): NameList [...]
)
```

```
class tensorbay.label.label_polyline.LabeledPolyline2D(points: Optional[Iterable[Iterable[float]]] =
                                                         None, *, category: Optional[str] = None,
                                                         attributes: Optional[Dict[str, Any]] = None,
                                                         instance: Optional[str] = None)
```

Bases: [tensorbay.geometry.point_list.PointList2D\[tensorbay.geometry.vector.Vector2D\]](#)

This class defines the concept of polyline2D label.

[LabeledPolyline2D](#) is the 2D polyline type of label, which is often used for CV tasks such as lane detection.

Parameters

- **points** – A list of 2D points representing the vertexes of the 2D polyline.
- **category** – The category of the label.
- **attributes** – The attributes of the label.
- **instance** – The instance id of the label.

category

The category of the label.

Type str

attributes

The attributes of the label.

Type Dict[str, Union[str, int, float, bool, List[Union[str, int, float, bool]]]]

instance

The instance id of the label.

Type str

Examples

```
>>> LabeledPolyline2D(
...     [(1, 2), (2, 4), (2, 1)],
...     category="example",
...     attributes={"key": "value"},
...     instance="123",
... )
LabeledPolyline2D [
  Vector2D(1, 2),
  Vector2D(2, 4),
  Vector2D(2, 1)
](
  (category): 'example',
  (attributes): {...},
  (instance): '123'
)
```

classmethod `loads(contents: Dict[str, Any])` → `tensorbay.label.label_polyline._T`

Loads a `LabeledPolyline2D` from a dict containing the information of the label.

Parameters `contents` – A dict containing the information of the 2D polyline label.

Returns The loaded `LabeledPolyline2D` object.

Examples

```
>>> contents = {
...     "polyline2d": [{ 'x': 1, 'y': 2}, { 'x': 2, 'y': 4}, { 'x': 2, 'y': 1}],
...     "category": "example",
...     "attributes": {"key": "value"},
...     "instance": "12345",
... }
>>> LabeledPolyline2D.loads(contents)
LabeledPolyline2D [
  Vector2D(1, 2),
  Vector2D(2, 4),
  Vector2D(2, 1)
](
  (category): 'example',
```

(continues on next page)

(continued from previous page)

```
(attributes): {...},
(instance): '12345'
)
```

dumps() → Dict[str, Any]

Dumps the current 2D polyline label into a dict.

Returns A dict containing all the information of the 2D polyline label.

Examples

```
>>> labeledpolyline2d = LabeledPolyline2D(
...     [(1, 2), (2, 4), (2, 1)],
...     category="example",
...     attributes={"key": "value"},
...     instance="123",
... )
>>> labeledpolyline2d.dumps()
{
    'category': 'example',
    'attributes': {'key': 'value'},
    'instance': '123',
    'polyline2d': [{'x': 1, 'y': 2}, {'x': 2, 'y': 4}, {'x': 2, 'y': 1}],
}
```

class tensorbay.label.label_polyline.**MultiPolyline2DSubcatalog**(*is_tracking: bool = 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** – A boolean value indicates whether the corresponding subcatalog contains tracking information.**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:
    Optional[Iterable[Iterable[float]]] =
    None, *, category: Optional[str] =
    None, attributes: Optional[Dict[str,
    Any]] = None, instance: Optional[str]
    = None)

Bases: tensorbay.geometry.point_list.MultiPointList2D[tensorbay.geometry.polyline.
Polyline2D]
```

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: Dict[str, Any]*) → tensorbay.label.label_polyline._T

Loads a LabeledMultiPolyline2D from a dict containing the information of the label.

Parameters **contents** – A dict containing the information of the 2D polyline label.

Returns The loaded *LabeledMultiPolyline2D* object.

Examples

```
>>> contents = {
...     "multiPolyline2d": [[{'x': 1, 'y': 1}, {'x': 1, 'y': 2}, {'x': 2, 'y': 2},
...                          {'x': 2, 'y': 3}, {'x': 3, 'y': 3}, {'x': 3, 'y': 5}]],
...     "category": "example",
...     "attributes": {"key": "value"},
...     "instance": "12345",
... }
>>> LabeledMultiPolyline2D.loads(contents)
LabeledMultiPolyline2D [
  Polyline2D [...]
  Polyline2D [...]
```

(continues on next page)

(continued from previous page)

```
](
    (category): 'example',
    (attributes): {...},
    (instance): '12345'
)
```

dumps() → Dict[str, Any]

Dumps the current 2D multiple polyline label into a dict.

Returns A dict containing all the information of the 2D polyline label.

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 transcribed 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 transcribed sentence type of label. which is often used for tasks such as automatic speech recognition.

class tensorbay.label.label_sentence.**SentenceSubcatalog**(*is_sample: bool = False, sample_rate: Optional[int] = None, lexicon: Optional[List[List[str]]] = None*)

Bases: *tensorbay.label.basic.SubcatalogBase*, *tensorbay.label.supports.AttributesMixin*

This class defines the subcatalog for audio transcribed sentence type of labels.

Parameters

- **is_sample** – A boolean value indicates whether time format is sample related.
- **sample_rate** – The number of samples of audio carried per second.
- **lexicon** – A list consists all of text and phone.

description

The description of the entire sentence subcatalog.

Type str

is_sample

A boolean 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.

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() → Dict[str, Any]

Dumps the information of this `SentenceSubcatalog` into a dict.

Returns A dict containing all information of this `SentenceSubcatalog`.

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: List[str]*) → None

Add lexemes to lexicon.

Parameters **lexemes** – A list consists of text and phone.

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: str, begin: Optional[float] = None, end: Optional[float] = 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** – The content of the word.
- **begin** – The begin time of the word in the audio.
- **end** – The end time of the word in the audio.

text

The content of the word.

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: Dict[str, Union[str, float]]) → tensorbay.label.label_sentence._T`
 Loads a Word from a dict containing the information of the word.

Parameters `contents` – A dict containing the information of the word

Returns The loaded `Word` object.

Examples

```
>>> contents = {"text": "Hello, World", "begin": 1, "end": 2}
>>> Word.loads(contents)
Word(
  (text): 'Hello, World',
  (begin): 1,
  (end): 2
)
```

dumps() → Dict[str, Union[str, float]]
 Dumps the current word into a dict.

Returns A dict containing all the information of the word

Examples

```
>>> word = Word(text="example", begin=1, end=2)
>>> word.dumps()
{'text': 'example', 'begin': 1, 'end': 2}
```

class `tensorbay.label.label_sentence.LabeledSentence(sentence: Optional[Iterable[tensorbay.label.label_sentence.Word]] = None, spell: Optional[Iterable[tensorbay.label.label_sentence.Word]] = None, phone: Optional[Iterable[tensorbay.label.label_sentence.Word]] = None, *, attributes: Optional[Dict[str, Any]] = None)`

Bases: `tensorbay.label.basic._LabelBase`

This class defines the concept of phonetic transcription lable.

`LabeledSentence` is the transcribed sentence type of label. which is often used for tasks such as automatic speech recognition.

Parameters

- **sentence** – A list of sentence.

- **spell** – A list of spell, only exists in Chinese language.
- **phone** – A list of phone.
- **attributes** – The attributes of the label.

sentence

The transcribed 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: Dict[str, Any]*) → `tensorbay.label.label_sentence._T`

Loads a `LabeledSentence` from a dict containing the information of the label.

Parameters **contents** – A dict containing the information of the sentence label.

Returns The loaded `LabeledSentence` object.

Examples

```

>>> contents = {
...     "sentence": [{"text": "qilshi2", "begin": 1, "end": 2}],
...     "spell": [{"text": "qil", "begin": 1, "end": 2}],
...     "phone": [{"text": "q", "begin": 1, "end": 2}],
...     "attributes": {"key": "value"},
... }
>>> LabeledSentence.loads(contents)
LabeledSentence(
  (sentence): [
    Word(
      (text): 'qilshi2',
      (begin): 1,
      (end): 2
    )
  ],
  (spell): [
    Word(
      (text): 'qil',
      (begin): 1,
      (end): 2
    )
  ],
  (phone): [
    Word(
      (text): 'q',
      (begin): 1,
      (end): 2
    )
  ],
  (attributes): {
    'key': 'value'
  }
)

```

dumps() → `Dict[str, Any]`

Dumps the current label into a dict.

Returns A dict containing all the information of the sentence label.

Examples

```
>>> sentence = [Word(text="qilshi2", begin=1, end=2)]
>>> spell = [Word(text="qil", 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': 'qilshi2', 'begin': 1, 'end': 2}],
  'spell': [{'text': 'qil', '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	explanation
<i>IsTrackingMixin</i>	a mixin class supporting tracking information of a subcatalog
<i>CategoriesMixin</i>	a mixin class supporting category information of a subcatalog
<i>AttributesMixin</i>	a mixin class supporting attribute information of a subcatalog

class tensorbay.label.supports.**CategoryInfo**(name: str, description: str = "")

Bases: *tensorbay.utility.name.NameMixin*

This class represents the information of a category, including category name and description.

Parameters

- **name** – The name of the category.
- **description** – The description of the category.

name

The name of the category.

description

The description of the category.

Type str

Examples

```
>>> CategoryInfo(name="example", description="This is an example")
CategoryInfo("example")
```

classmethod `loads(contents: Dict[str, str]) → tensorbay.label.supports._T`

Loads a `CategoryInfo` from a dict containing the category.

Parameters `contents` – A dict containing the information of the category.

Returns The loaded `CategoryInfo` object.

Examples

```
>>> contents = {"name": "example", "description": "This is an exmaple"}
>>> CategoryInfo.loads(contents)
CategoryInfo("example")
```

classmethod `dumps() → Dict[str, str]`

Dumps the `CategoryInfo` into a dict.

Returns A dict containing the information in the `CategoryInfo`.

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: str, category_id: int, description: str = "")`

Bases: `tensorbay.label.supports.CategoryInfo`

This class represents the information of a category, including name, id and description.

Parameters

- **name** – The name of the category.
- **category_id** – The id of the category.
- **description** – The description of the category.

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: int, *, names: Optional[Iterable[str]] = None,  
                                              skeleton: Optional[Iterable[Iterable[int]]] = None,  
                                              visible: Optional[str] = None, parent_categories:  
                                              Union[None, str, Iterable[str]] = None, description: str =  
                                              "")
```

Bases: [tensorbay.utility.repr.ReprMixin](#), [tensorbay.utility.attr.AttrsMixin](#)

This class defines the structure of a set of keypoints.

Parameters

- **number** – The number of the set of keypoints.
- **names** – All the names of the keypoints.
- **skeleton** – The skeleton of the keypoints indicating which keypoint should connect with another.
- **visible** – The visible type of the keypoints, can only be ‘BINARY’ or ‘TERNARY’. It determines the range of the [Keypoint2D.v](#).
- **parent_categories** – The parent categories of the keypoints.
- **description** – The description of the keypoints.

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: Dict[str, Any]) → tensorbay.label.supports._T`

Loads a KeypointsInfo from a dict containing the information of the keypoints.

Parameters `contents` – A dict containing all the information of the set of keypoints.

Returns The loaded *KeypointsInfo* object.

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() → Dict[str, Any]

Dumps all the keypoint information into a dict.

Returns A dict containing all the information of the keypoint.

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: bool = False*)

Bases: `tensorbay.utility.attr.AttrsMixin`

A mixin class supporting tracking information of a subcatalog.

Parameters `is_tracking` – Whether the Subcatalog contains tracking information.

is_tracking

Whether the Subcatalog contains tracking information.

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() → `Dict[str, int]`

Return the dict containing the conversion from category to index.

Returns A dict containing the conversion from category to index.

get_index_to_category() → `Dict[int, str]`

Return the dict containing the conversion from index to category.

Returns A dict containing the conversion from index to category.

add_category(*name: str, description: str = ""*) → `None`

Add a category to the Subcatalog.

Parameters

- **name** – The name of the category.
- **description** – The description of the category.

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() → `Dict[str, int]`

Return the dict containing the conversion from category name to category id.

Returns A dict containing the conversion from category name to category id.

get_index_to_category() → `Dict[int, str]`

Return the dict containing the conversion from category id to category name.

Returns A dict containing the conversion from category id to category name.

add_category(*name: str, category_id: int, description: str = ""*) → `None`

Add a category to the Subcatalog.

Parameters

- **name** – The name of the category.
- **category_id** – The id of the category.
- **description** – The description of the category.

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: str, *, type_: Union[str, None, Type[Optional[Union[list, bool, int, float, str]]], Iterable[Union[str, None, Type[Optional[Union[list, bool, int, float, str]]]]] = "", enum: Optional[Iterable[Optional[Union[str, float, bool]]]] = None, minimum: Optional[float] = None, maximum: Optional[float] = None, items: Optional[tensorbay.label.attributes.Items] = None, parent_categories: Union[None, str, Iterable[str]] = None, description: str = ""*) → `None`

Add an attribute to the Subcatalog.

Parameters

- **name** – The name of the attribute.
- **type** – The type of the attribute value, could be a single type or multi-types. The type must be within the followings: - array - boolean - integer - number - string - null - instance

- **enum** – All the possible values of an enumeration attribute.
- **minimum** – The minimum value of number type attribute.
- **maximum** – The maximum value of number type attribute.
- **items** – The items inside array type attributes.
- **parent_categories** – The parent categories of the attribute.
- **description** – The description of the attributes.

1.20.5 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 coefficients. 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: Optional[float] = None, fy: Optional[float] = None,
                                                cx: Optional[float] = None, cy: Optional[float] =
                                                None, skew: float = 0, *, matrix:
                                                Optional[Union[Sequence[Sequence[float]],
                                                                numpy.ndarray]] = 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** – The x axis focal length expressed in pixels.
- **fy** – The y axis focal length expressed in pixels.
- **cx** – The x coordinate of the so called principal point that should be in the center of the image.
- **cy** – The y coordinate of the so called principal point that should be in the center of the image.
- **skew** – It causes shear distortion in the projected image.
- **matrix** – A 3x3 Sequence of camera matrix.

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.

Examples

```
>>> matrix = [[1, 3, 3],
...           [0, 2, 4],
...           [0, 0, 1]]
```

Initialization 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
)
```

Initialization 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: Dict[str, float]*) → tensorbay.sensor.intrinsics._T

Loads CameraMatrix from a dict containing the information of the camera matrix.

Parameters **contents** – A dict containing the information of the camera matrix.

Returns A *CameraMatrix* instance contains the information from the contents dict.

Examples

```
>>> contents = {
...     "fx": 2,
...     "fy": 6,
...     "cx": 4,
...     "cy": 7,
...     "skew": 3
... }
>>> camera_matrix = CameraMatrix.loads(contents)
>>> camera_matrix
CameraMatrix(
  (fx): 2,
  (fy): 6,
  (cx): 4,
  (cy): 7,
  (skew): 3
)
```

dumps() → Dict[str, float]

Dumps the camera matrix into a dict.

Returns A dict containing the information of the camera matrix.

Examples

```
>>> camera_matrix.dumps()
{'fx': 1, 'fy': 2, 'cx': 3, 'cy': 4, 'skew': 3}
```

as_matrix() → numpy.ndarray

Return the camera matrix as a 3x3 numpy array.

Returns A 3x3 numpy array representing the camera matrix.

Examples

```
>>> numpy_array = camera_matrix.as_matrix()
>>> numpy_array
array([[1., 3., 3.],
       [0., 4., 4.],
       [0., 0., 1.]])
```

project(point: Sequence[float]) → *tensorbay.geometry.vector.Vector2D*

Project a point to the pixel coordinates.

Parameters point – A Sequence containing the coordinates of the point to be projected.

Returns The pixel coordinates.

Raises TypeError – When the dimension of the input point is neither two nor three.

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: float*)
 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`, ...

Raises `TypeError` – When tangential and radial distortion is not provided to initialize class.

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: Dict[str, float]*) → `tensorbay.sensor.intrinsics._T`
 Loads `DistortionCoefficients` from a dict containing the information.

Parameters *contents* – A dict containing distortion coefficients of a camera.

Returns A `DistortionCoefficients` instance containing information from the *contents* dict.

Examples

```
>>> contents = {
...     "p1": 1,
...     "p2": 2,
...     "k1": 3,
...     "k2": 4
... }
>>> distortion_coefficients = DistortionCoefficients.loads(contents)
>>> distortion_coefficients
DistortionCoefficients(
  (p1): 1,
  (p2): 2,
```

(continues on next page)

(continued from previous page)

```
(k1): 3,
(k2): 4
)
```

dumps() → Dict[str, float]

Dumps the distortion coefficients into a dict.

Returns A dict containing the information of distortion coefficients.

Examples

```
>>> distortion_coefficients.dumps()
{'p1': 1, 'p2': 2, 'k1': 3, 'k2': 4}
```

distort(point: Sequence[float], is_fisheye: bool = False) → *tensorbay.geometry.vector.Vector2D*

Add distortion to a point.

Parameters

- **point** – A Sequence containing the coordinates of the point to be distorted.
- **is_fisheye** – Whether the sensor is fisheye camera, default is False.

Raises **TypeError** – When the dimension of the input point is neither two nor three.**Returns** Distorted 2d point.

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: Optional[float] = None, fy: Optional[float] =
None, cx: Optional[float] = None, cy:
Optional[float] = None, skew: float = 0, *,
camera_matrix:
Optional[Union[Sequence[Sequence[float]],
numpy.ndarray]] = None, **kwargs: float)
```

Bases: *tensorbay.utility.repr.ReprMixin*, *tensorbay.utility.attr.AttrsMixin*

CameraIntrinsics represents camera intrinsics.

Camera intrinsic parameters including camera matrix and distortion coefficients. They describe the mapping of the scene in front of the camera to the pixels in the final image.

Parameters

- **fx** – The x axis focal length expressed in pixels.
- **fy** – The y axis focal length expressed in pixels.
- **cx** – The x coordinate of the so called principal point that should be in the center of the image.
- **cy** – The y coordinate of the so called principal point that should be in the center of the image.
- **skew** – It causes shear distortion in the projected image.
- **camera_matrix** – A 3x3 Sequence of the camera matrix.
- ****kwargs** – Float values to initialize *DistortionCoefficients*.

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,
```

(continues on next page)

(continued from previous page)

```

...     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: Dict[str, Dict[str, float]])` → `tensorbay.sensor.intrinsics._T`

Loads `CameraIntrinsics` from a dict containing the information.

Parameters `contents` – A dict containig camera matrix and distortion coefficients.

Returns A `CameraIntrinsics` instance containing information from the contents dict.

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(
  (camera_matrix): CameraMatrix(
    (fx): 1,
    (fy): 2,
    (cx): 3,

```

(continues on next page)

(continued from previous page)

```

        (cy): 4,
        (skew): 0
    ),
    (distortion_coefficients): DistortionCoefficients(
        (p1): 1,
        (p2): 2,
        (k1): 3,
        (k2): 4
    )
)

```

dumps() → Dict[str, Dict[str, float]]

Dumps the camera intrinsics into a dict.

Returns A dict containing camera intrinsics.

Examples

```

>>> camera_intrinsics.dumps()
{'cameraMatrix': {'fx': 1, 'fy': 2, 'cx': 3, 'cy': 4, 'skew': 3},
 'distortionCoefficients': {'p1': 5, 'k1': 6}}

```

set_camera_matrix(*fx: Optional[float] = None, fy: Optional[float] = None, cx: Optional[float] = None, cy: Optional[float] = None, skew: float = 0, *, matrix: Optional[Union[Sequence[Sequence[float]], numpy.ndarray]] = None*) → None

Set camera matrix of the camera intrinsics.

Parameters

- **fx** – The x axis focal length expressed in pixels.
- **fy** – The y axis focal length expressed in pixels.
- **cx** – The x coordinate of the so called principal point that should be in the center of the image.
- **cy** – The y coordinate of the so called principal point that should be in the center of the image.
- **skew** – It causes shear distortion in the projected image.
- **matrix** – Camera matrix in 3x3 sequence.

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
  )
)

```

(continues on next page)

(continued from previous page)

```

    ),
    (distortion_coefficients): DistortionCoefficients(
        (p1): 1,
        (p2): 2,
        (k1): 3,
        (k2): 4
    )
)

```

set_distortion_coefficients(**kwargs: float) → None

Set distortion coefficients of the camera intrinsics.

Parameters ****kwargs** – Contains p1, p2, ..., k1, k2, ...

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: Sequence[float], is_fisheye: bool = False) → *tensorbay.geometry.vector.Vector2D*

Project a point to the pixel coordinates.

If distortion coefficients are provided, distort the point before projection.

Parameters

- **point** – A Sequence containing coordinates of the point to be projected.
- **is_fisheye** – Whether the sensor is fisheye camera, default is False.

Returns The coordinates on the pixel plane where the point is projected to.

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: str)`

Bases: `tensorbay.utility.name.NameMixin`, `tensorbay.utility.type.TypeMixin[tensorbay.sensor.sensor.SensorType]`

Sensor defines the concept of sensor.

Sensor includes name, description, translation and rotation.

Parameters `name` – *Sensor*'s name.

Raises **TypeError** – Can not instantiate abstract class *Sensor*.

extrinsics

The translation and rotation of the sensor.

Type `tensorbay.geometry.transform.Transform3D`

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

Loads a Sensor from a dict containing the sensor information.

Parameters `contents` – A dict containing name, description and sensor extrinsics.

Returns A *Sensor* instance containing the information from the contents dict.

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)
    )
)
```

dumps() → `Dict[str, Any]`

Dumps the sensor into a dict.

Returns A dict containing the information of the sensor.

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: Iterable[float] = (0, 0, 0), rotation: Union[Iterable[float], quaternion.quaternion] = (1, 0, 0, 0), *, matrix: Optional[Union[Sequence[Sequence[float]], numpy.ndarray]] = None) → None

Set the extrinsics of the sensor.

Parameters

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

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: float, y: float, z: float) → None

Set the translation of the sensor.

Parameters

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

Examples

```
>>> sensor.set_translation(x=2, y=3, z=4)
>>> sensor
Lidar("Lidar1")(
  (extrinsics): Transform3D(
    (translation): Vector3D(2, 3, 4),
    ...
  )
)
```

set_rotation(w: *Optional[float] = None*, x: *Optional[float] = None*, y: *Optional[float] = None*, z: *Optional[float] = None*, *, quaternion: *Optional[quaternion.quaternion] = None*) → None
Set the rotation of the sensor.

Parameters

- **w** – The w componet of the roation quaternion.
- **x** – The x componet of the roation quaternion.
- **y** – The y componet of the roation quaternion.
- **z** – The z componet of the roation quaternion.
- **quaternion** – Numpy quaternion representing the rotation.

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: str)

Bases: [tensorbay.utility.name.NameMixin](#), [tensorbay.utility.type.TypeMixin](#)[[tensorbay.sensor.sensor.SensorType](#)]

Lidar defines the concept of lidar.

[Lidar](#) is a kind of sensor for measuring distances by illuminating the target with laser light and measuring the reflection.

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)
  )
)
```

class `tensorbay.sensor.sensor.Radar(name: str)`
 Bases: `tensorbay.utility.name.NameMixin`, `tensorbay.utility.type.TypeMixin[tensorbay.sensor.sensor.SensorType]`

Radar defines the concept of radar.

Radar is a detection system that uses radio waves to determine the range, angle, or velocity of objects.

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)
  )
)
```

class `tensorbay.sensor.sensor.Camera(name: str)`
 Bases: `tensorbay.utility.name.NameMixin`, `tensorbay.utility.type.TypeMixin[tensorbay.sensor.sensor.SensorType]`

Camera defines the concept of camera.

Camera includes name, description, translation, rotation, cameraMatrix and distortionCoefficients.

extrinsics

The translation and rotation of the camera.

Type `tensorbay.geometry.transform.Transform3D`

intrinsics

The camera matrix and distortion coefficients of the camera.

Type `tensorbay.sensor.intrinsics.CameraIntrinsics`

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: Dict[str, Any]) → tensorbay.sensor.sensor._T`

Loads a Camera from a dict containing the camera information.

Parameters `contents` – A dict containing name, description, extrinsics and intrinsics.

Returns A *Camera* instance containing information from contents dict.

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},
...     },
... }

```

(continues on next page)

(continued from previous page)

```

... }
>>> 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() → Dict[str, Any]

Dumps the camera into a dict.

Returns A dict containing name, description, extrinsics and intrinsics.

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: Optional[float] = None, fy: Optional[float] = None, cx: Optional[float] = None, cy: Optional[float] = None, skew: float = 0, *, matrix: Optional[Union[Sequence[Sequence[float]], numpy.ndarray]] = None) → None

Set camera matrix.

Parameters

- **fx** – The x axis focal length expressed in pixels.

- **fy** – The y axis focal length expressed in pixels.
- **cx** – The x coordinate of the so called principal point that should be in the center of the image.
- **cy** – The y coordinate of the so called principal point that should be in the center of the image.
- **skew** – It causes shear distortion in the projected image.
- **matrix** – Camera matrix in 3x3 sequence.

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: float) → None

Set distortion coefficients.

Parameters ****kwargs** – Float values to set distortion coefficients.

Raises **ValueError** – When intrinsics is not set yet.

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: str)

Bases: [tensorbay.utility.name.NameMixin](#), [tensorbay.utility.type.TypeMixin](#)[[tensorbay.sensor.sensor.SensorType](#)]

FisheyeCamera defines the concept of fisheye camera.

Fisheye camera is an ultra wide-angle lens that produces strong visual distortion intended to create a wide panoramic or hemispherical image.

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)
  )
)
```

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: List[Dict[str, Any]]) → tensorbay.sensor.sensor._T

Loads a [Sensors](#) instance from the given contents.

Parameters contents – 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": {
      "cameraMatrix": {
        "fx": <float>
        "fy": <float>
        "cx": <float>
        "cy": <float>
      }
    }
  },
  ...
]
```

--- only for cameras

(continues on next page)

(continued from previous page)

```

        "skew": <float>
    }
    "distortionCoefficients": {
        "k1": <float>
        "k2": <float>
        "p1": <float>
        "p2": <float>
        ...
    }
},
"description": <str>
},
...
]

```

Returns The loaded *Sensors* instance.

dumps() → List[Dict[str, Any]]

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": {
            "cameraMatrix": {
                "fx": <float>
                "fy": <float>
                "cx": <float>
                "cy": <float>
                "skew": <float>
            }
            "distortionCoefficients": {
                "k1": <float>
                "k2": <float>
                "p1": <float>
                "p2": <float>
            }
        }
    }
]

```

(continues on next page)

(continued from previous page)

```

        ...
    },
    "description": <str>
},
...
]

```

1.20.6 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: bool, key: Union[str, None, Callable[[str], str]],
                                   default: Any, error_message: Optional[str], loader:
                                   Optional[Callable[[Any], Any]], dumper: Optional[Callable[[Any],
                                   Any]])
```

Bases: object

A class to identify attr fields.

Parameters

- **is_dynamic** – Whether attr is a dynamic attr.
- **key** – Display value of the attr in contents.
- **default** – Default value of the attr.
- **error_message** – The custom error message of the attr.
- **loader** – The custom loader of the attr.
- **dumper** – The custom dumper of the attr.

```
class tensorbay.utility.attr.BaseField(key: Optional[str])
```

Bases: object

A class to identify fields of base class.

Parameters **key** – Display value of the attr.

```
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: bool = False, key: Union[str, None, Callable[[str], str]] =  
    <function <lambda>>, default: Any = Ellipsis, error_message: Optional[str] =  
    None, loader: Optional[Callable[[Any], Any]] = None, dumper:  
    Optional[Callable[[Any], Any]] = None) → Any
```

Return an instance to identify attr fields.

Parameters

- **is_dynamic** – Determine if this is a dynamic attr.
- **key** – Display value of the attr in contents.
- **default** – Default value of the attr.
- **error_message** – The custom error message of the attr.
- **loader** – The custom loader of the attr.
- **dumper** – The custom dumper of the attr.

Raises [AttrError](#) – Dynamic attr cannot have default value.

Returns A [Field](#) instance containing all attr fields.

```
tensorbay.utility.attr.attr_base(key: Optional[str] = None) → Any
```

Return an instance to identify base class fields.

Parameters **key** – Display value of the attr.

Returns A [BaseField](#) instance containing all base class fields.

```
tensorbay.utility.attr.upper(name: str) → str
```

Convert the name value to uppercase.

Parameters **name** – name of the attr.

Returns The uppercase value.

```
tensorbay.utility.attr.camel(name: str) → str
```

Convert the name value to camelcase.

Parameters **name** – name of the attr.

Returns The camelcase value.

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: Type[tensorbay.utility.common._T], contents: Any)  
    → tensorbay.utility.common._T
```

A common method for loading an object from a dict or a list of dict.

Parameters

- **object_class** – The class of the object to be loaded.
- **contents** – The information of the object in a dict or a list of dict.

Returns The loaded object.

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: tensorbay.utility.common._CallableWithoutReturnValue) → tensorbay.utility.common._CallableWithoutReturnValue`

The decorator to add threading lock for methods.

Parameters `func` – The method needs to add threading lock.

Returns The method with theading locked.

`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: str, description: str = "")`

Bases: `tensorbay.utility.attr.AttrsMixin`, `tensorbay.utility.repr.ReprMixin`

A mixin class for instance which has immutable name and mutable description.

Parameters

- **name** – Name of the class.
- **description** – Description of the class.

name

Name of the class.

class `tensorbay.utility.name.NameList(values: Iterable[tensorbay.utility.name._T] = ())`

Bases: `tensorbay.utility.user.UserSequence[tensorbay.utility.name._T]`

`NameList` is a list of named elements, supports searching the element by its name.

keys() → `Tuple[str, ...]`

Get all element names.

Returns A tuple containing all elements names.

append(*value: tensorbay.utility.name._T*) → `None`

Append element to the end of the `NameList`.

Parameters `value` – Element to be appended to the `NameList`.

Raises **KeyError** – When the name of the appending object already exists in the `NameList`.

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: tensorbay.utility.name._T*) → None
Store element in name sorted list.

Parameters **value** – The element needs to be added to the list.

Raises **KeyError** – If the name of the added value exists in the list.

keys() → Tuple[str, ...]
Get all element names.

Returns A tuple containing all elements names.

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, **kws)
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: tensorbay.utility.type._T)
    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, **kws)
    Bases: Sequence[tensorbay.utility.user._T], tensorbay.utility.repr.ReprMixin
```

`UserSequence` is a user-defined wrapper around sequence objects.

index(value: tensorbay.utility.user._T, start: int = 0, stop: int = 9223372036854775807) → int
Return the first index of the value.

Parameters

- **value** – The value to be found.
- **start** – The start index of the subsequence.
- **stop** – The end index of the subsequence.

Returns The First index of value.

count(value: tensorbay.utility.user._T) → int
Return the number of occurrences of value.

Parameters `value` – The value to be counted the number of occurrences.

Returns The number of occurrences of value.

```
class tensorbay.utility.user.UserMutableSequence(*args, **kws)
    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: int, value: tensorbay.utility.user._T) → None
Insert object before index.

Parameters

- **index** – Position of the mutable sequence.
- **value** – Element to be inserted into the mutable sequence.

append(value: tensorbay.utility.user._T) → None
Append object to the end of the mutable sequence.

Parameters `value` – Element to be appended to the mutable sequence.

clear() → None

Remove all items from the mutable sequence.

extend(values: Iterable[*tensorbay.utility.user._T*]) → None

Extend mutable sequence by appending elements from the iterable.

Parameters values – Elements to be Extended into the mutable sequence.

reverse() → None

Reverse the items of the mutable sequence in place.

pop(index: int = - 1) → *tensorbay.utility.user._T*

Return the item at index (default last) and remove it from the mutable sequence.

Parameters index – Position of the mutable sequence.

Returns Element to be removed from the mutable sequence.

remove(value: *tensorbay.utility.user._T*) → None

Remove the first occurrence of value.

Parameters value – Element to be removed from the mutable sequence.

class *tensorbay.utility.user.UserMapping*(*args, **kws)

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() → AbstractSet[Tuple[*tensorbay.utility.user._K*, *tensorbay.utility.user._V*]]

Return a new view of the (key, value) pairs in dict.

Returns The (key, value) pairs in dict.

keys() → AbstractSet[*tensorbay.utility.user._K*]

Return a new view of the keys in dict.

Returns The keys in dict.

values() → ValuesView[*tensorbay.utility.user._V*]

Return a new view of the values in dict.

Returns The values in dict.

class *tensorbay.utility.user.UserMutableMapping*(*args, **kws)

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() → None

Remove all items from the mutable mapping object.

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() → *Tuple[tensorbay.utility.user._K, tensorbay.utility.user._V]*
 Remove and return a (key, value) pair as a tuple.

Pairs are returned in LIFO (last-in, first-out) order.

Returns A (key, value) pair as a tuple.

setdefault(key: *tensorbay.utility.user._K*, default: *Optional[tensorbay.utility.user._V] = None*) → *tensorbay.utility.user._V*
 Set the value of the item with the specified key.

If the key is in the dict, return the corresponding value. If not, insert the key with a value of default and return default.

Parameters

- **key** – The key for dict, which can be any immutable type.
- **default** – The value to be set if the key is not in the dict.

Returns The value for key if it is in the dict, else default.

update(__m: *Mapping[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.

1.20.7 tensorbay.exception

TensorBay cutoms exceptions.

The class hierarchy for TensorBay custom exceptions is:

```
+-- TensorBayException
+-- ClientError
+-- StatusError
+-- DatasetTypeError
+-- FrameError
+-- ResponseError
```

(continues on next page)

```

    +-- AccessDeniedError
    +-- InvalidParamsError
    +-- NameConflictError
    +-- RequestParamsMissingError
    +-- ResourceNotExistError
    +-- ResponseSystemError
    +-- UnauthorizedError
+-- UtilityError
    +-- AttrError
    +-- TBRNError
    +-- OpenDatasetError
        +-- NoFileError
        +-- FileStructureError

```

exception `tensorbay.exception.TensorBayException`

Bases: `Exception`

This is the base class for TensorBay custom exceptions.

exception `tensorbay.exception.ClientError`

Bases: `tensorbay.exception.TensorBayException`

This is the base class for custom exceptions in TensorBay client module.

exception `tensorbay.exception.StatusError`(*is_draft: Optional[bool] = None, message: Optional[str] = None*)

Bases: `tensorbay.exception.ClientError`

This class defines the exception for illegal status.

Parameters

- **is_draft** – Whether the status is draft.
- **message** – The error message.

exception `tensorbay.exception.DatasetTypeError`(*dataset_name: str, is_fusion: bool*)

Bases: `tensorbay.exception.ClientError`

This class defines the exception for incorrect type of the requested dataset.

Parameters

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

exception `tensorbay.exception.FrameError`(*message: str*)

Bases: `tensorbay.exception.ClientError`

This class defines the exception for incorrect frame id.

Parameters **message** – The error message.

exception `tensorbay.exception.OperationError`(*message: str*)

Bases: `tensorbay.exception.ClientError`

This class defines the exception for incorrect operation.

Parameters **message** – The error message.

exception `tensorbay.exception.ResponseError(response: requests.models.Response)`

Bases: `tensorbay.exception.ClientError`

This class defines the exception for post response error.

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

response

The response of the request.

exception `tensorbay.exception.AccessDeniedError(response: requests.models.Response)`

Bases: `tensorbay.exception.ResponseError`

This class defines the exception for access denied response error.

exception `tensorbay.exception.InvalidParamsError(response: Optional[requests.models.Response] = None, *, param_name: Optional[str] = None, param_value: Optional[str] = None)`

Bases: `tensorbay.exception.ResponseError`

This class defines the exception for invalid parameters response error.

Parameters

- **response** – The response of the request.
- **param_name** – The name of the invalid parameter.
- **param_value** – The value of the invalid parameter.

response

The response of the request.

exception `tensorbay.exception.NameConflictError(response: Optional[requests.models.Response] = None, *, resource: Optional[str] = None, identification: Optional[Union[int, str]] = None)`

Bases: `tensorbay.exception.ResponseError`

This class defines the exception for name conflict response error.

Parameters

- **response** – The response of the request.
- **resource** – The type of the conflict resource.
- **identification** – The identification of the conflict resource.

response

The response of the request.

exception `tensorbay.exception.RequestParamsMissingError(response: requests.models.Response)`

Bases: `tensorbay.exception.ResponseError`

This class defines the exception for request parameters missing response error.

exception `tensorbay.exception.ResourceNotExistError(response: Optional[requests.models.Response] = None, *, resource: Optional[str] = None, identification: Optional[Union[int, str]] = None)`

Bases: `tensorbay.exception.ResponseError`

This class defines the exception for resource not existing response error.

Parameters

- **response** – The response of the request.
- **resource** – The type of the conflict resource.
- **identification** – The identification of the conflict resource.
- **response** – The response of the request.

exception `tensorbay.exception.ResponseSystemError(response: requests.models.Response)`
Bases: `tensorbay.exception.ResponseError`

This class defines the exception for system response error.

exception `tensorbay.exception.UnauthorizedError(response: requests.models.Response)`
Bases: `tensorbay.exception.ResponseError`

This class defines the exception for unauthorized response error.

exception `tensorbay.exception.OpenDatasetError`
Bases: `tensorbay.exception.TensorBayException`

This is the base class for custom exceptions in TensorBay opendataset module.

exception `tensorbay.exception.NoFileError(pattern: str)`
Bases: `tensorbay.exception.OpenDatasetError`

This class defines the exception for no matching file found in the opendataset directory.

Parameters `pattern` – Glob pattern.

exception `tensorbay.exception.FileStructureError(message: str)`
Bases: `tensorbay.exception.OpenDatasetError`

This class defines the exception for incorrect file structure in the opendataset directory.

Parameters `message` – The error message.

exception `tensorbay.exception.ModuleImportError(module_name: str, package_name: Optional[str] = None)`
Bases: `tensorbay.exception.OpenDatasetError`, `ModuleNotFoundError`

This class defines the exception for import error of optional module in opendataset module.

Parameters

- **module_name** – The name of the optional module.
- **package_name** – The package name of the optional module.

exception `tensorbay.exception.TBRNError(message: str)`
Bases: `tensorbay.exception.TensorBayException`

This class defines the exception for invalid TBRN.

Parameters `message` – The error message.

exception `tensorbay.exception.UtilityError`
Bases: `tensorbay.exception.TensorBayException`

This is the base class for custom exceptions in TensorBay utility module.

exception `tensorbay.exception.AttrError`
Bases: `tensorbay.exception.UtilityError`

This class defines the exception for dynamic attr have default value.

1.20.8 tensorbay.opendataset

tensorbay.opendataset.AADB.loader

`tensorbay.opendataset.AADB.loader.AADB(path: str) → tensorbay.dataset.dataset.Dataset`
Load the AADB to TensorBay.

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` – the root directory of the dataset.

Returns Loaded *Dataset* instance.

tensorbay.opendataset.AnimalPose.loader

`tensorbay.opendataset.AnimalPose.loader.AnimalPose5(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of the 5 Categories Animal-Pose dataset.

The file structure should be like:

```
<path>
  keypoint_image_part1/
    cat/
      2007_000549.jpg
      2007_000876.jpg
      ...
    ...
  PASCAL2011_animal_annotation/
    cat/
      2007_000549_1.xml
      2007_000876_1.xml
      2007_000876_2.xml
      ...
    ...
  animalpose_image_part2/
    cat/
      ca1.jpeg
      ca2.jpeg
      ...
    ...
  animalpose_anno2/
    cat/
      ca1.xml
      ca2.xml
    ...
```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.AnimalPose.loader.AnimalPose7(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of 7 Categories Animal-Pose dataset.

The file structure should be like:

```
<path>
  bndbox_image/
    antelope/
      Img-77.jpg
      ...
    ...
  bndbox_anno/
    antelope.json
    ...
```

Parameters `path` – The root directory of the dataset.

Returns loaded *Dataset* object.

`tensorbay.opendataset.AnimalsWithAttributes2.loader`

`tensorbay.opendataset.AnimalsWithAttributes2.loader.AnimalsWithAttributes2(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of the 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` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.BDD100K.loader`

This file defines the BDD100K dataloader.

`tensorbay.opendataset.BDD100K.loader.BDD100K(path: str) → tensorbay.dataset.dataset.Dataset`
Load BDD100K Dataset to Tensorbay.

The file structure should be like:

```
<path>
  bdd100k_images_100k/
    images/
```

(continues on next page)

(continued from previous page)

```

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` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.BSTLD.loader`

`tensorbay.opendataset.BSTLD.loader.BSTLD(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of the *BSTLD* 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

```

Parameters `path` – The root directory of the dataset.

Raises *ModuleImportError* – When the module “yaml” can not be found.

Returns Loaded *Dataset* instance.

tensorbay.opendataset.BioIDFace.loader

`tensorbay.opendataset.BioIDFace.loader.BioIDFace(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of [The 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` – The root directory of the dataset.

Returns Loaded [Dataset](#) instance.

tensorbay.opendataset.CACD.loader

`tensorbay.opendataset.CACD.loader.CACD(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of [Cross-Age Celebrity Dataset \(CACD\)](#) dataset.

The file structure should be like:

```
<path>
  CACD2000/
    14_Aaron_Johnson_0001.jpg
    ...
  celebrity2000.mat
```

Parameters `path` – The root directory of the dataset.

Returns Loaded [Dataset](#) instance.

tensorbay.opendataset.CADC.loader

`tensorbay.opendataset.CADC.loader.CADC(path: str) → tensorbay.dataset.dataset.FusionDataset`
Dataloader of the [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
```

(continues on next page)

(continued from previous page)

```

...
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` – The root directory of the dataset.

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

tensorbay.opendataset.CCPD.loader

tensorbay.opendataset.CCPD.loader.CCPD(`path: str`) → `tensorbay.dataset.dataset.Dataset`

Dataloader of CCPD open 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/

```

(continues on next page)

(continued from previous page)

```

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` – The root directory of the dataset.

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

Return type Loaded

`tensorbay.opendataset.CCPD.loader.CCPDGreen(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of CCPDGreen open dataset.

The file structure should be like:

```

<path>
  ccpd_green/
    train/
    test/
    val/

```

Parameters `path` – The root directory of the dataset.

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

Return type Loaded

`tensorbay.opendataset.COVIDChestXRay.loader`

`tensorbay.opendataset.COVIDChestXRay.loader.COVIDChestXRay(path: str) →`

`tensorbay.dataset.dataset.Dataset`

Dataloader of 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

```

(continues on next page)

(continued from previous page)

```

...
volumes/
    coronacases_org_001.nii.gz
....
metadata.csv
...

```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.COVID_CT.loader`

`tensorbay.opendataset.COVID_CT.loader.COVID_CT(path: str) → tensorbay.dataset.dataset.Dataset`
 Dataloader of the **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.jpg
      ...
      91%1.jpg
      102.png
      ...
      2341.png

```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

tensorbay.opendataset.CarConnection.loader

tensorbay.opendataset.CarConnection.loader.**CarConnection**(path: str) → *tensorbay.dataset.dataset.Dataset*

Dataloader of The Car Connection Picture dataset.

The file structure should be like:

```
<path>
  <imagename>.jpg
  ...
```

Parameters path – The root directory of the dataset.

Returns Loaded *Dataset* instance.

tensorbay.opendataset.CoinImage.loader

tensorbay.opendataset.CoinImage.loader.**CoinImage**(path: str) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the Coin Image dataset.

The file structure should be like:

```
<path>
  classes.csv
  <imagename>.png
  ...
```

Parameters path – The root directory of the dataset.

Returns Loaded *Dataset* instance.

tensorbay.opendataset.CompCars.loader

tensorbay.opendataset.CompCars.loader.**CompCars**(path: str) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the 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>/
```

(continues on next page)

(continued from previous page)

```

                                <image name>.txt
                                ...
                                ...
                                ...
                                ...
                                misc/
                                  attributes.txt
                                  car_type.mat
                                  make_model_name.mat
                                train_test_split/
                                  classification/
                                    train.txt
                                    test.txt

```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.DeepRoute.loader`

`tensorbay.opendataset.DeepRoute.loader.DeepRoute(path: str) → tensorbay.dataset.dataset.Dataset`
 Dataloader of the *DeepRoute Open Dataset*.

The file structure should be like:

```

<path>
  pointcloud/
    00001.bin
    00002.bin
    ...
    10000.bin
  groundtruth/
    00001.txt
    00002.txt
    ...
    10000.txt

```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.DogsVsCats.loader`

`tensorbay.opendataset.DogsVsCats.loader.DogsVsCats(path: str) → tensorbay.dataset.dataset.Dataset`
 Dataloader of the *Dogs vs Cats* dataset.

The file structure should be like:

```

<path>
  train/
    cat.0.jpg
    ...

```

(continues on next page)

(continued from previous page)

```
dog.0.jpg
...
test/
  1000.jpg
  1001.jpg
  ...
```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.DownsamplingImagenet.loader`

`tensorbay.opendataset.DownsamplingImagenet.loader.DownsamplingImagenet(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of the *Downsampling 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` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.Elpy.loader`

`tensorbay.opendataset.Elpy.loader.Elpy(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of the *elpy* dataset.

The file structure should be like:

```
<path>
  labels.csv
  images/
    cell10001.png
    ...
```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

tensorbay.opendataset.FLIC.loader

`tensorbay.opendataset.FLIC.loader.FLIC(path: str) → tensorbay.dataset.dataset.Dataset`
 Dataloader of the [FLIC](#) dataset.

The folder structure should be like:

```
<path>
  exampls.mat
  images/
    2-fast-2-furious-00003571.jpg
    ...
```

Parameters `path` – The root directory of the dataset.

Raises `ModuleImportError` – When the module “scipy” can not be found.

Returns Loaded `Dataset` instance.

tensorbay.opendataset.FSDD.loader

`tensorbay.opendataset.FSDD.loader.FSDD(path: str) → tensorbay.dataset.dataset.Dataset`
 Dataloader of the [Free Spoken Digit](#) dataset.

The file structure should be like:

```
<path>
  recordings/
    0_george_0.wav
    0_george_1.wav
    ...
```

Parameters `path` – The root directory of the dataset.

Returns Loaded `Dataset` instance.

tensorbay.opendataset.Flower.loader

`tensorbay.opendataset.Flower.loader.Flower17(path: str) → tensorbay.dataset.dataset.Dataset`
 Dataloader of the [17 Category Flower](#) dataset.

The dataset are 3 separate splits. The results in the paper are averaged over the 3 splits. We just use (trn1, val1, tst1) to split it.

The file structure should be like:

```
<path>
  jpg/
    image_0001.jpg
    ...
  datasplits.mat
```

Parameters `path` – The root directory of the dataset.

Raises `ModuleImportError` – When the module “scipy” can not be found.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.Flower.loader.Flower102(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of the 102 Category Flower dataset.

The file structure should be like:

```
<path>
  jpg/
    image_000001.jpg
    ...
  imagelabels.mat
  setid.mat
```

Parameters `path` – The root directory of the dataset.

Raises *ModuleImportError* – When the module “scipy” can not be found.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.HalpeFullBody.loader`

`tensorbay.opendataset.HalpeFullBody.loader.HalpeFullBody(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of the 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_000000001.jpg
        ...
      val2017/
        000000000139.jpg
        ...
```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.HardHatWorkers.loader`

`tensorbay.opendataset.HardHatWorkers.loader.HardHatWorkers(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of the Hard Hat Workers dataset.

The file structure should be like:

```
<path>
  annotations/
    hard_hat_workers0.xml
    ...
  images/
    hard_hat_workers0.png
    ...
```

Parameters `path` – The root directory of the dataset.

Returns Loaded `Dataset` instance.

`tensorbay.opendataset.HeadPoseImage.loader`

`tensorbay.opendataset.HeadPoseImage.loader.HeadPoseImage(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of the `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` – The root directory of the dataset.

Returns Loaded `Dataset` instance.

`tensorbay.opendataset.ImageEmotion.loader`

`tensorbay.opendataset.ImageEmotion.loader.ImageEmotionAbstract(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of the `Image Emotion-abstract` dataset.

The file structure should be like:

```
<path>
  ABSTRACT_groundTruth.csv
  abstract_xxxx.jpg
  ...
```

Parameters `path` – The root directory of the dataset.

Returns Loaded `Dataset` instance.

`tensorbay.opendataset.ImageEmotion.loader.ImageEmotionArtphoto(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of the [Image Emotion-art Photo](#) dataset.

The file structure should be like:

```
<path>
  <filename>.jpg
  ...
```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.JHU_CROWD.loader`

`tensorbay.opendataset.JHU_CROWD.loader.JHU_CROWD(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of the [JHU-CROWD++](#) dataset.

The file structure should be like:

```
<path>
  train/
    images/
      0000.jpg
      ...
    gt/
      0000.txt
      ...
    image_labels.txt
  test/
  val/
```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.KenyanFood.loader`

`tensorbay.opendataset.KenyanFood.loader.KenyanFoodOrNonfood(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of the [Kenyan Food or Nonfood](#) dataset.

The file structure should be like:

```
<path>
  images/
    food/
      236171947206673742.jpg
      ...
    nonfood/
      168223407.jpg
      ...
  data.csv
```

(continues on next page)

(continued from previous page)

```
split.py
test.txt
train.txt
```

Parameters `path` – The root directory of the dataset.

Returns Loaded `Dataset` instance.

`tensorbay.opendataset.KenyanFood.loader.KenyanFoodType(path: str) →`
`tensorbay.dataset.dataset.Dataset`

Dataloader of the 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` – The root directory of the dataset.

Returns Loaded `Dataset` instance.

tensorbay.opendataset.KylbergTexture.loader

tensorbay.opendataset.KylbergTexture.loader.**KylbergTexture**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the [Kylberg Texture](#) dataset.

The file structure should be like:

```
<path>
  originalPNG/
    <imagename>.png
    ...
  withoutRotateAll/
    <imagename>.png
    ...
  RotateAll/
    <imagename>.png
    ...
```

Parameters **path** – The root directory of the dataset.

Returns Loaded [Dataset](#) instance.

tensorbay.opendataset.LISATrafficLight.loader

tensorbay.opendataset.LISATrafficLight.loader.**LISATrafficLight**(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the [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/
```

(continues on next page)

(continued from previous page)

```

nightSequence1/nightSequence1/
nightSequence2/nightSequence2/
nightTrain/nightTrain/
    nightClip1/
    nightClip2/
    ...
    nightClip5/

```

Parameters `path` – The root directory of the dataset.

Returns Loaded `Dataset` instance.

Raises `FileStructureError` – When frame number is discontinuous.

`tensorbay.opendataset.LISATrafficSign.loader`

`tensorbay.opendataset.LISATrafficSign.loader.LISATrafficSign(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of the [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/

```

(continues on next page)

(continued from previous page)

```

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/
    frameAnnotations-MVI_0121.MOV_annotations/
negatives/
    negativePics/
    negatives.dat
tools/
    evaluateDetections.py
    extractAnnotations.py
    mergeAnnotationFiles.py
    splitAnnotationFiles.py

```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.LeedsSportsPose.loader`

`tensorbay.opendataset.LeedsSportsPose.loader.LeedsSportsPose(path: str) → tensorbay.dataset.dataset.Dataset`

Dataloader of the *Leeds Sports Pose* dataset.

The folder structure should be like:

```

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

```

Parameters `path` – The root directory of the dataset.

Raises *ModuleNotFoundError* – When the module “scipy” can not be found.

Returns Loaded *Dataset* instance.

tensorbay.opendataset.NeolixOD.loader

`tensorbay.opendataset.NeolixOD.loader.NeolixOD(path: str) → tensorbay.dataset.dataset.Dataset`
 Dataloader of the [Neolix OD](#) dataset.

The file structure should be like:

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

Parameters `path` – The root directory of the dataset.

Returns Loaded `Dataset` instance.

tensorbay.opendataset.Newsgroups20.loader

`tensorbay.opendataset.Newsgroups20.loader.Newsgroups20(path: str) → tensorbay.dataset.dataset.Dataset`
 Dataloader of the [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/
```

(continues on next page)

(continued from previous page)

```
20news-bydate-train/  
20_newsgroups/
```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.NightOwls.loader`

`tensorbay.opendataset.NightOwls.loader.NightOwls(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of the *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` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.RP2K.loader`

`tensorbay.opendataset.RP2K.loader.RP2K(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of the *RP2K* dataset.

The file structure of RP2K looks like:

```
<path>  
  all/  
    test/  
      <catagory>/  
        <image_name>.jpg  
        ...  
      ...  
    train/  
      <catagory>/  
        <image_name>.jpg  
        ...  
    ...
```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.THCHS30.loader`

`tensorbay.opendataset.THCHS30.loader.THCHS30(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of the *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` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

`tensorbay.opendataset.THUCNews.loader`

`tensorbay.opendataset.THUCNews.loader.THUCNews(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of the *THUCNews* dataset.

The folder structure should be like:

```
<path>
  <category>/
    0.txt
    1.txt
    2.txt
    3.txt
    ...
  <category>/
    ...
```

Parameters `path` – The root directory of the dataset.

Returns Loaded *Dataset* instance.

tensorbay.opendataset.TLR.loader

tensorbay.opendataset.TLR.loader.TLR(*path: str*) → *tensorbay.dataset.dataset.Dataset*
Dataloader of the 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* – The root directory of the dataset.

Returns Loaded *Dataset* instance.

tensorbay.opendataset.UAVDT.loader

tensorbay.opendataset.UAVDT.loader.UAVDT(*path: str*) → *tensorbay.dataset.dataset.Dataset*
Dataloader of the 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* – The root directory of the dataset.

Returns Loaded *Dataset* instance.

tensorbay.opendataset.VOC2012ActionClassification.loader

tensorbay.opendataset.VOC2012ActionClassification.loader.VOC2012ActionClassification(*path: str*) →

→
ten-
sor-
bay.dataset.dataset.Dataset

Dataloader of the ‘VOC2012ActionClassification’_ dataset.

The file structure should be like:

```
<path>
  Annotations/
    <image_name>.xml
    ...
  JPEGImages/
    <image_name>.jpg
    ...
  ImageSets/
    Action/
      train.txt
      trainval.txt
      val.txt
      ...
    ...
  ...
```

Parameters *path* – The root directory of the dataset.

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

Return type Loaded

tensorbay.opendataset.VOC2012Detection.loader

tensorbay.opendataset.VOC2012Detection.loader.VOC2012Detection(*path: str*) → *tensorbay.dataset.dataset.Dataset*

Dataloader of the ‘VOC2012Detection’_ dataset.

The file structure should be like:

```
<path>
  Annotations/
    <image_name>.xml
    ...
  JPEGImages/
    <image_name>.jpg
    ...
  ImageSets/
    Main/
      train.txt
      trainval.txt
      val.txt
```

(continues on next page)

(continued from previous page)

```
...  
...  
...
```

Parameters `path` – The root directory of the dataset.

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

Return type Loaded

`tensorbay.opendataset.WIDER_FACE.loader`

`tensorbay.opendataset.WIDER_FACE.loader.WIDER_FACE(path: str) → tensorbay.dataset.dataset.Dataset`
Dataloader of the `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/  
  WIDER_val/  
    ...  
  WIDER_test/  
    ...  
  wider_face_split/  
    wider_face_train_bbx_gt.txt  
    wider_face_val_bbx_gt.txt
```

Parameters `path` – The root directory of the dataset.

Returns Loaded `Dataset` instance.

`tensorbay.opendataset._utility`

OpenDataset utility code.

`tensorbay.opendataset._utility.coco(path: str) → tensorbay.opendataset._utility.coco.COCO`
Parse the coco-like label files.

Parameters `path` – The label directory of the dataset.

Returns

A dict containing four dicts:

dicts	keys	values
images	image id	information of image files
annotations	annotation id	annotations
categories	category id	all categories
images_annotations_map	image id	annotation id

`tensorbay.opendataset._utility.glob(pathname: str, *, recursive: bool = False) → List[str]`

Return a sorted list of paths matching a pathname pattern.

The pattern may contain simple shell-style wildcards a la `fnmatch`. However, unlike `fnmatch`, filenames starting with a dot are special cases that are not matched by `*` and `?` patterns.

Parameters

- **pathname** – The pathname pattern.
- **recursive** – If recursive is true, the pattern `**` will match any files and zero or more directories and subdirectories.

Returns A sorted list of paths matching a pathname pattern.

Raises `NoFileError` – When there is no file matching the given pathname pattern.

tensorbay.opendataset.nuScenes.loader

`tensorbay.opendataset.nuScenes.loader.nuScenes(path: str) → tensorbay.dataset.dataset.FusionDataset`

Dataloader of the `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/
    sweeps/
      CAM_BACK/
      CAM_BACK_LEFT/
      CAM_BACK_RIGHT/
      CAM_FRONT/
```

(continues on next page)

(continued from previous page)

```
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` – The root directory of the dataset.

Returns Loaded *FusionDataset* instance.

PYTHON MODULE INDEX

t

- tensorbay.client.cloud_storage, 112
- tensorbay.client.dataset, 113
- tensorbay.client.gas, 118
- tensorbay.client.lazy, 121
- tensorbay.client.log, 123
- tensorbay.client.requests, 125
- tensorbay.client.segment, 128
- tensorbay.client.status, 131
- tensorbay.client.struct, 132
- tensorbay.client.version, 136
- tensorbay.dataset.data, 140
- tensorbay.dataset.dataset, 142
- tensorbay.dataset.frame, 146
- tensorbay.dataset.segment, 144
- tensorbay.exception, 255
- tensorbay.geometry.box, 147
- tensorbay.geometry.keypoint, 154
- tensorbay.geometry.point_list, 156
- tensorbay.geometry.polygon, 157
- tensorbay.geometry.polyline, 160
- tensorbay.geometry.transform, 163
- tensorbay.geometry.vector, 166
- tensorbay.label.attributes, 170
- tensorbay.label.basic, 175
- tensorbay.label.catalog, 176
- tensorbay.label.label, 178
- tensorbay.label.label_box, 179
- tensorbay.label.label_classification, 187
- tensorbay.label.label_keypoints, 189
- tensorbay.label.label_mask, 194
- tensorbay.label.label_polygon, 202
- tensorbay.label.label_polyline, 212
- tensorbay.label.label_sentence, 218
- tensorbay.label.supports, 224
- tensorbay.opendataset._utility, 282
- tensorbay.opendataset.AADB.loader, 259
- tensorbay.opendataset.AnimalPose.loader, 259
- tensorbay.opendataset.AnimalsWithAttributes2.loader, 260
- tensorbay.opendataset.BDD100K.loader, 260
- tensorbay.opendataset.BioIDFace.loader, 262
- tensorbay.opendataset.BSTLD.loader, 261
- tensorbay.opendataset.CACD.loader, 262
- tensorbay.opendataset.CADC.loader, 262
- tensorbay.opendataset.CarConnection.loader, 266
- tensorbay.opendataset.CCPD.loader, 263
- tensorbay.opendataset.CoinImage.loader, 266
- tensorbay.opendataset.CompCars.loader, 266
- tensorbay.opendataset.COVID_CT.loader, 265
- tensorbay.opendataset.COVIDChestXRay.loader, 264
- tensorbay.opendataset.DeepRoute.loader, 267
- tensorbay.opendataset.DogsVsCats.loader, 267
- tensorbay.opendataset.DownsamplingImaginet.loader, 268
- tensorbay.opendataset.Elpy.loader, 268
- tensorbay.opendataset.FLIC.loader, 269
- tensorbay.opendataset.Flower.loader, 269
- tensorbay.opendataset.FSDD.loader, 269
- tensorbay.opendataset.HalpeFullBody.loader, 270
- tensorbay.opendataset.HardHatWorkers.loader, 270
- tensorbay.opendataset.HeadPoseImage.loader, 271
- tensorbay.opendataset.ImageEmotion.loader, 271
- tensorbay.opendataset.JHU_CROWD.loader, 272
- tensorbay.opendataset.KenyanFood.loader, 272
- tensorbay.opendataset.KylbergTexture.loader, 274
- tensorbay.opendataset.LeedsSportsPose.loader, 276
- tensorbay.opendataset.LISATrafficLight.loader, 274
- tensorbay.opendataset.LISATrafficSign.loader, 275
- tensorbay.opendataset.NeolixOD.loader, 277
- tensorbay.opendataset.Newsgroups20.loader, 277
- tensorbay.opendataset.NightOwls.loader, 278
- tensorbay.opendataset.nuScenes.loader, 283

- `tensorbay.opendataset.RP2K.loader`, 278
- `tensorbay.opendataset.THCHS30.loader`, 279
- `tensorbay.opendataset.THUCNews.loader`, 279
- `tensorbay.opendataset.TLR.loader`, 280
- `tensorbay.opendataset.UAVDT.loader`, 280
- `tensorbay.opendataset.VOC2012ActionClassification.loader`, 281
- `tensorbay.opendataset.VOC2012Detection.loader`, 281
- `tensorbay.opendataset.WIDER_FACE.loader`, 282
- `tensorbay.sensor.intrinsics`, 230
- `tensorbay.sensor.sensor`, 239
- `tensorbay.utility.attr`, 249
- `tensorbay.utility.common`, 250
- `tensorbay.utility.name`, 251
- `tensorbay.utility.repr`, 252
- `tensorbay.utility.type`, 252
- `tensorbay.utility.user`, 253

A

- AADB() (in module *tensorbay.opendataset.AADB.loader*), 259
- AccessDeniedError, 111, 257
- add() (*tensorbay.utility.name.SortedNameList* method), 251
- add_attribute() (tensorbay.label.supports.AttributesMixin method), 229
- add_category() (tensorbay.label.supports.CategoriesMixin method), 228
- add_category() (tensorbay.label.supports.MaskCategoriesMixin method), 229
- add_keypoints() (tensorbay.label.label_keypoints.Keypoints2DSubcatalog method), 191
- add_segment() (*tensorbay.dataset.dataset.DatasetBase* method), 144
- alias (tensorbay.client.dataset.DatasetClientBase attribute), 113
- all_attributes (tensorbay.label.label_mask.InstanceMask attribute), 199
- all_attributes (tensorbay.label.label_mask.InstanceMaskBase attribute), 198
- all_attributes (tensorbay.label.label_mask.PanopticMask attribute), 200
- all_attributes (tensorbay.label.label_mask.PanopticMaskBase attribute), 198
- all_attributes (tensorbay.label.label_mask.RemoteInstanceMask attribute), 201
- all_attributes (tensorbay.label.label_mask.RemotePanopticMask attribute), 202
- all_attributes (tensorbay.label.label_mask.RemoteSemanticMask attribute), 200
- all_attributes (tensorbay.label.label_mask.SemanticMask attribute), 198
- all_attributes (tensorbay.label.label_mask.SemanticMaskBase attribute), 198
- all_category_ids (tensorbay.label.label_mask.PanopticMask attribute), 200
- all_category_ids (tensorbay.label.label_mask.PanopticMaskBase attribute), 198
- allowed_retry_methods (tensorbay.client.requests.Config attribute), 125
- allowed_retry_status (tensorbay.client.requests.Config attribute), 125
- AnimalPose5() (in module *tensorbay.opendataset.AnimalPose.loader*), 259
- AnimalPose7() (in module *tensorbay.opendataset.AnimalPose.loader*), 260
- AnimalsWithAttributes2() (in module *tensorbay.opendataset.AnimalsWithAttributes2.loader*), 260
- append() (*tensorbay.client.lazy.PagingList* method), 123
- append() (*tensorbay.utility.name.NameList* method), 251
- append() (*tensorbay.utility.user.UserMutableSequence* method), 253
- append_lexicon() (tensorbay.label.label_sentence.SentenceSubcatalog method), 220
- area() (*tensorbay.geometry.box.Box2D* method), 150
- area() (*tensorbay.geometry.polygon.Polygon* method), 158
- as_matrix() (*tensorbay.geometry.transform.Transform3D* method), 165
- as_matrix() (*tensorbay.sensor.intrinsics.CameraMatrix* method), 232
- attr() (in module *tensorbay.utility.attr*), 250
- attr_base() (in module *tensorbay.utility.attr*), 250
- AttrError, 258

- [AttributeInfo](#) (class in `tensorbay.label.attributes`), 172
[attributes](#) (`tensorbay.label.label_box.Box2DSubcatalog` attribute), 180
[attributes](#) (`tensorbay.label.label_box.Box3DSubcatalog` attribute), 184
[attributes](#) (`tensorbay.label.label_box.LabeledBox2D` attribute), 181
[attributes](#) (`tensorbay.label.label_box.LabeledBox3D` attribute), 185
[attributes](#) (`tensorbay.label.label_classification.Classification` attribute), 189
[attributes](#) (`tensorbay.label.label_classification.ClassificationSubcatalog` attribute), 187
[attributes](#) (`tensorbay.label.label_keypoints.Keypoints2DSubcatalog` attribute), 190
[attributes](#) (`tensorbay.label.label_keypoints.LabeledKeypoints2D` attribute), 193
[attributes](#) (`tensorbay.label.label_mask.InstanceMaskSubcatalog` attribute), 196
[attributes](#) (`tensorbay.label.label_mask.PanopticMaskSubcatalog` attribute), 197
[attributes](#) (`tensorbay.label.label_mask.SemanticMaskSubcatalog` attribute), 195
[attributes](#) (`tensorbay.label.label_polygon.LabeledMultiPolygon` attribute), 208
[attributes](#) (`tensorbay.label.label_polygon.LabeledPolygon` attribute), 206
[attributes](#) (`tensorbay.label.label_polygon.LabeledRLE` attribute), 210
[attributes](#) (`tensorbay.label.label_polygon.MultiPolygonSubcatalog` attribute), 204
[attributes](#) (`tensorbay.label.label_polygon.PolygonSubcatalog` attribute), 203
[attributes](#) (`tensorbay.label.label_polygon.RLESubcatalog` attribute), 205
[attributes](#) (`tensorbay.label.label_polyline.LabeledMultiPolyline2D` attribute), 217
[attributes](#) (`tensorbay.label.label_polyline.LabeledPolyline2D` attribute), 214
[attributes](#) (`tensorbay.label.label_polyline.MultiPolyline2DSubcatalog` attribute), 215
[attributes](#) (`tensorbay.label.label_polyline.Polyline2DSubcatalog` attribute), 212
[attributes](#) (`tensorbay.label.label_sentence.LabeledSentence` attribute), 222
[attributes](#) (`tensorbay.label.label_sentence.SentenceSubcatalog` attribute), 219
[attributes](#) (`tensorbay.label.supports.AttributesMixin` attribute), 229
[AttributesMixin](#) (class in `tensorbay.label.supports`), 229
[AttrsMixin](#) (class in `tensorbay.utility.attr`), 249
[AuthData](#) (class in `tensorbay.dataset.data`), 142
- ## B
- [BaseField](#) (class in `tensorbay.utility.attr`), 249
[BDD100K](#) (in module `tensorbay.opendataset.BDD100K.loader`), 260
[begin](#) (`tensorbay.label.label_sentence.Word` attribute), 220
[BioIDFace](#) (in module `tensorbay.opendataset.BioIDFace.loader`), 262
[bounds](#) (`tensorbay.geometry.point_list.MultiPointList2D` method), 157
[bounds](#) (`tensorbay.geometry.point_list.PointList2D` method), 156
[Box2D](#) (class in `tensorbay.geometry.box`), 147
[Box2DSubcatalog](#) (class in `tensorbay.label.label_box`), 179
[Box3D](#) (class in `tensorbay.geometry.box`), 151
[Box3DSubcatalog](#) (class in `tensorbay.label.label_box`), 183
[br](#) (`tensorbay.geometry.box.Box2D` property), 149
[Branch](#) (class in `tensorbay.client.struct`), 135
[BSTLD](#) (in module `tensorbay.opendataset.BSTLD.loader`), 261
[CACD](#) (in module `tensorbay.opendataset.CACD.loader`), 262
[CADC](#) (in module `tensorbay.opendataset.CADC.loader`), 262
[camel](#) (in module `tensorbay.utility.attr`), 250
[Camera](#) (class in `tensorbay.sensor.sensor`), 243
[camera_matrix](#) (`tensorbay.sensor.intrinsics.CameraIntrinsics` attribute), 235
[CameraIntrinsics](#) (class in `tensorbay.sensor.intrinsics`), 234
[cameraMatrix](#) (class in `tensorbay.sensor.intrinsics`), 230
[CarConnection](#) (in module `tensorbay.opendataset.CarConnection.loader`), 266
[catalog](#) (class in `tensorbay.label.catalog`), 176
[catalog](#) (`tensorbay.dataset.dataset.DatasetBase` attribute), 143
[categories](#) (`tensorbay.label.label_box.Box2DSubcatalog` attribute), 179
[categories](#) (`tensorbay.label.label_box.Box3DSubcatalog` attribute), 183
[categories](#) (`tensorbay.label.label_classification.ClassificationSubcatalog` attribute), 187
[categories](#) (`tensorbay.label.label_keypoints.Keypoints2DSubcatalog` attribute), 189
[categories](#) (`tensorbay.label.label_mask.InstanceMaskSubcatalog` attribute), 196
[categories](#) (`tensorbay.label.label_mask.PanopticMaskSubcatalog` attribute), 197

categories (*tensorbay.label.label_mask.SemanticMaskSubcatalog* attribute), 195
categories (*tensorbay.label.label_polygon.MultiPolygonSubcatalog* attribute), 204
categories (*tensorbay.label.label_polygon.PolygonSubcatalog* attribute), 203
categories (*tensorbay.label.label_polygon.RLESubcatalog* attribute), 205
categories (*tensorbay.label.label_polyline.MultiPolyline2DSubcatalog* attribute), 215
categories (*tensorbay.label.label_polyline.Polyline2DSubcatalog* attribute), 212
categories (*tensorbay.label.supports.CategoriesMixin* attribute), 228
categories (*tensorbay.label.supports.MaskCategoriesMixin* attribute), 229
CategoriesMixin (class in *tensorbay.label.supports*), 228
category (*tensorbay.label.label_box.LabeledBox2D* attribute), 181
category (*tensorbay.label.label_box.LabeledBox3D* attribute), 185
category (*tensorbay.label.label_classification.Classification* attribute), 188
category (*tensorbay.label.label_keypoints.LabeledKeypoints2D* attribute), 193
category (*tensorbay.label.label_polygon.LabeledMultiPolygon* attribute), 208
category (*tensorbay.label.label_polygon.LabeledPolygon* attribute), 206
category (*tensorbay.label.label_polygon.LabeledRLE* attribute), 210
category (*tensorbay.label.label_polyline.LabeledMultiPolyline2D* attribute), 217
category (*tensorbay.label.label_polyline.LabeledPolyline2D* attribute), 213
category_delimiter (*tensorbay.label.label_box.Box2DSubcatalog* attribute), 180
category_delimiter (*tensorbay.label.label_box.Box3DSubcatalog* attribute), 184
category_delimiter (*tensorbay.label.label_classification.ClassificationSubcatalog* attribute), 187
category_delimiter (*tensorbay.label.label_keypoints.Keypoints2DSubcatalog* attribute), 190
category_delimiter (*tensorbay.label.label_mask.InstanceMaskSubcatalog* attribute), 196
category_delimiter (*tensorbay.label.label_mask.PanopticMaskSubcatalog* attribute), 197
category_delimiter (*tensorbay.label.label_mask.SemanticMaskSubcatalog* attribute), 195
category_delimiter (*tensorbay.label.label_polygon.MultiPolygonSubcatalog* attribute), 204
category_delimiter (*tensorbay.label.label_polygon.PolygonSubcatalog* attribute), 203
category_delimiter (*tensorbay.label.label_polygon.RLESubcatalog* attribute), 205
category_delimiter (*tensorbay.label.label_polyline.MultiPolyline2DSubcatalog* attribute), 215
category_delimiter (*tensorbay.label.label_polyline.Polyline2DSubcatalog* attribute), 212
category_delimiter (*tensorbay.label.supports.CategoriesMixin* attribute), 228
category_delimiter (*tensorbay.label.supports.MaskCategoriesMixin* attribute), 229
category_delimiter (*tensorbay.label.supports.MaskCategoryInfo* attribute), 225
category_id (*tensorbay.label.supports.MaskCategoryInfo* attribute), 225
CategoryInfo (class in *tensorbay.label.supports*), 224
CCPD() (in module *tensorbay.opendataset.CCPD.loader*), 263
CCPDGreen() (in module *tensorbay.opendataset.CCPD.loader*), 264
check_authority_for_commit() (*tensorbay.client.status.Status* method), 131
check_authority_for_draft() (*tensorbay.client.status.Status* method), 131
checkout() (*tensorbay.client.status.Status* method), 131
checkout() (*tensorbay.client.version.VersionControlClient* method), 137
Classification (class in *tensorbay.label.label_classification*), 188
ClassificationSubcatalog (class in *tensorbay.label.label_classification*), 187
clear() (*tensorbay.utility.user.UserMutableMapping* method), 254
clear() (*tensorbay.utility.user.UserMutableSequence* method), 253
Client (class in *tensorbay.client.requests*), 126
ClientError, 111, 256
close_draft() (*tensorbay.client.version.VersionControlClient* method), 138
CloudClient (class in *tensorbay.client.cloud_storage*), 112
coco() (in module *tensorbay.opendataset._utility*), 282

CoinImage() (in module *tensorbay.opendataset.CoinImage.loader*), 266
 Commit (class in *tensorbay.client.struct*), 134
 commit() (*tensorbay.client.version.VersionControlClient* method), 137
 commit_id (*tensorbay.client.status.Status* property), 131
 common_loads() (in module *tensorbay.utility.common*), 250
 CompCars() (in module *tensorbay.opendataset.CompCars.loader*), 266
 Config (class in *tensorbay.client.requests*), 125
 copy_data() (*tensorbay.client.segment.SegmentClient* method), 129
 copy_segment() (*tensorbay.client.dataset.DatasetClient* method), 114
 copy_segment() (*tensorbay.client.dataset.FusionDatasetClient* method), 116
 count() (*tensorbay.client.lazy.PagingList* method), 123
 count() (*tensorbay.utility.user.UserSequence* method), 253
 COVID_CT() (in module *tensorbay.opendataset.COVID_CT.loader*), 265
 COVIDChestXRay() (in module *tensorbay.opendataset.COVIDChestXRay.loader*), 264
 create_auth_dataset() (*tensorbay.client.gas.GAS* method), 119
 create_branch() (*tensorbay.client.version.VersionControlClient* method), 138
 create_dataset() (*tensorbay.client.gas.GAS* method), 118
 create_draft() (*tensorbay.client.version.VersionControlClient* method), 137
 create_segment() (*tensorbay.client.dataset.DatasetClient* method), 114
 create_segment() (*tensorbay.client.dataset.FusionDatasetClient* method), 116
 create_segment() (*tensorbay.dataset.dataset.Dataset* method), 144
 create_segment() (*tensorbay.dataset.dataset.FusionDataset* method), 144
 create_tag() (*tensorbay.client.version.VersionControlClient* method), 139
 cx (*tensorbay.sensor.intrinsics.CameraMatrix* attribute), 231
 cy (*tensorbay.sensor.intrinsics.CameraMatrix* attribute),

231

D

Data (class in *tensorbay.dataset.data*), 140
 data (*tensorbay.client.lazy.LazyItem* attribute), 121
 DataBase (class in *tensorbay.dataset.data*), 140
 Dataset (class in *tensorbay.dataset.dataset*), 144
 dataset_id (*tensorbay.client.dataset.DatasetClientBase* attribute), 113
 dataset_id (*tensorbay.client.version.VersionControlClient* property), 137
 DatasetBase (class in *tensorbay.dataset.dataset*), 143
 DatasetClient (class in *tensorbay.client.dataset*), 114
 DatasetClientBase (class in *tensorbay.client.dataset*), 113
 DatasetTypeError, 111, 256
 DeepRoute() (in module *tensorbay.opendataset.DeepRoute.loader*), 267
 delete_branch() (*tensorbay.client.version.VersionControlClient* method), 139
 delete_data() (*tensorbay.client.segment.SegmentClientBase* method), 128
 delete_dataset() (*tensorbay.client.gas.GAS* method), 120
 delete_segment() (*tensorbay.client.dataset.DatasetClientBase* method), 114
 delete_sensor() (*tensorbay.client.segment.FusionSegmentClient* method), 130
 delete_tag() (*tensorbay.client.version.VersionControlClient* method), 139
 description (*tensorbay.label.attributes.AttributeInfo* attribute), 173
 description (*tensorbay.label.basic.SubcatalogBase* attribute), 175
 description (*tensorbay.label.label_box.Box2DSubcatalog* attribute), 179
 description (*tensorbay.label.label_box.Box3DSubcatalog* attribute), 183
 description (*tensorbay.label.label_classification.ClassificationSubcatalog* attribute), 187
 description (*tensorbay.label.label_keypoints.Keypoints2DSubcatalog* attribute), 189
 description (*tensorbay.label.label_mask.InstanceMaskSubcatalog* attribute), 196
 description (*tensorbay.label.label_mask.PanopticMaskSubcatalog* attribute), 197
 description (*tensorbay.label.label_mask.SemanticMaskSubcatalog* attribute), 194

`description` (*tensorbay.label.label_polygon.MultiPolygonSubcatalog* method), 162
 attribute), 204 `dumps()` (*tensorbay.geometry.transform.Transform3D*
`description` (*tensorbay.label.label_polygon.PolygonSubcatalog* method), 164
 attribute), 202 `dumps()` (*tensorbay.geometry.vector.Vector2D* method),
`description` (*tensorbay.label.label_polygon.RLESubcatalog* method), 168
 attribute), 205 `dumps()` (*tensorbay.geometry.vector.Vector3D* method),
`description` (*tensorbay.label.label_polyline.MultiPolyline2DSubcatalog* method), 176
 attribute), 215 `dumps()` (*tensorbay.label.attributes.AttributeInfo*
`description` (*tensorbay.label.label_polyline.Polyline2DSubcatalog* method), 174
 attribute), 212 `dumps()` (*tensorbay.label.attributes.Items* method), 172
`description` (*tensorbay.label.label_sentence.SentenceSubcatalog* method), 175
 attribute), 218 `dumps()` (*tensorbay.label.basic.SubcatalogBase*
`description` (*tensorbay.label.supports.CategoryInfo* *attribute*), 224 `dumps()` (*tensorbay.label.catalog.Catalog* method), 177
 attribute), 224 `dumps()` (*tensorbay.label.label.Label* method), 179
`description` (*tensorbay.label.supports.KeypointsInfo* *attribute*), 226 `dumps()` (*tensorbay.label.label_box.LabeledBox2D*
 attribute), 226 method), 183
`description` (*tensorbay.label.supports.MaskCategoryInfo* *attribute*), 225 `dumps()` (*tensorbay.label.label_box.LabeledBox3D*
 attribute), 225 method), 186
`distort()` (*tensorbay.sensor.intrinsics.DistortionCoefficients* method), 234 `dumps()` (*tensorbay.label.label_keypoints.Keypoints2DSubcatalog*
 method), 234 method), 192
`distortion_coefficients` (*tensorbay.sensor.intrinsics.CameraIntrinsics* *attribute*), 235 `dumps()` (*tensorbay.label.label_keypoints.LabeledKeypoints2D*
 attribute), 235 method), 194
`DistortionCoefficients` (class in *tensorbay.sensor.intrinsics*), 233 `dumps()` (*tensorbay.label.label_polygon.LabeledMultiPolygon*
 method), 209
`do()` (*tensorbay.client.requests.Client* method), 126 `dumps()` (*tensorbay.label.label_polygon.LabeledPolygon*
 method), 207
`DogsVsCats()` (in module *tensorbay.opendataset.DogsVsCats.loader*), 267 `dumps()` (*tensorbay.label.label_polygon.LabeledRLE*
 method), 211
`DownsampledImagenet()` (in module *tensorbay.opendataset.DownsampledImagenet.loader*), 268 `dumps()` (*tensorbay.label.label_polyline.LabeledMultiPolyline2D*
 method), 218
 `dumps()` (*tensorbay.label.label_polyline.LabeledPolyline2D*
 method), 215
`Draft` (class in *tensorbay.client.struct*), 135 `dumps()` (*tensorbay.label.label_sentence.LabeledSentence*
 method), 223
`draft_number` (*tensorbay.client.status.Status* property), 131 `dumps()` (*tensorbay.label.label_sentence.SentenceSubcatalog*
 method), 219
`dump_request_and_response()` (in module *tensorbay.client.log*), 123 `dumps()` (*tensorbay.label.label_sentence.Word* method),
 221
`dumps()` (*tensorbay.client.struct.Commit* method), 135 `dumps()` (*tensorbay.label.supports.CategoryInfo*
 method), 225
`dumps()` (*tensorbay.client.struct.Draft* method), 136 `dumps()` (*tensorbay.label.supports.KeypointsInfo*
 method), 227
`dumps()` (*tensorbay.client.struct.TeamInfo* method), 132 `dumps()` (*tensorbay.sensor.intrinsics.CameraIntrinsics*
 method), 237
`dumps()` (*tensorbay.client.struct.UserInfo* method), 133 `dumps()` (*tensorbay.sensor.intrinsics.CameraMatrix*
 method), 232
`dumps()` (*tensorbay.dataset.dataset.Notes* method), 143 `dumps()` (*tensorbay.sensor.intrinsics.DistortionCoefficients*
 method), 234
`dumps()` (*tensorbay.geometry.box.Box2D* method), 150 `dumps()` (*tensorbay.sensor.sensor.Camera* method), 245
`dumps()` (*tensorbay.geometry.box.Box3D* method), 153 `dumps()` (*tensorbay.sensor.sensor.Sensor* method), 240
`dumps()` (*tensorbay.geometry.keypoint.Keypoint2D* method), 155 `dumps()` (*tensorbay.sensor.sensor.Sensors* method), 248
`dumps()` (*tensorbay.geometry.point_list.MultiPointList2D* method), 157
`dumps()` (*tensorbay.geometry.point_list.PointList2D* method), 156
`dumps()` (*tensorbay.geometry.polygon.MultiPolygon* method), 159
`dumps()` (*tensorbay.geometry.polygon.RLE* method), 160
`dumps()` (*tensorbay.geometry.polyline.MultiPolyline2D*

E

`Elpv()` (in module *tensorbay.opendataset.Elpy.loader*),

268
 end (*tensorbay.label.label_sentence.Word* attribute), 220
 enum (*tensorbay.label.attributes.AttributeInfo* attribute), 173
 enum (*tensorbay.label.attributes.Items* attribute), 171
 enum (*tensorbay.utility.type.TypeMixin* property), 252
 EqMixin (*class* in *tensorbay.utility.common*), 251
 extend() (*tensorbay.client.lazy.PagingList* method), 123
 extend() (*tensorbay.utility.user.UserMutableSequence* method), 254

extrinsics (*tensorbay.sensor.sensor.Camera* attribute), 243

extrinsics (*tensorbay.sensor.sensor.Sensor* attribute), 240

F

Field (*class* in *tensorbay.utility.attr*), 249

FileStructureError, 112, 258

FisheyeCamera (*class* in *tensorbay.sensor.sensor*), 246

FLIC() (*in module tensorbay.opendataset.FLIC.loader*), 269

Flower102() (*in module tensorbay.opendataset.Flower.loader*), 270

Flower17() (*in module tensorbay.opendataset.Flower.loader*), 269

Frame (*class* in *tensorbay.dataset.frame*), 146

FrameError, 111, 256

from_data() (*tensorbay.client.lazy.LazyItem* class method), 121

from_page() (*tensorbay.client.lazy.LazyItem* class method), 121

from_response_body() (*tensorbay.dataset.data.RemoteData* class method), 141

from_response_body() (*tensorbay.dataset.frame.Frame* class method), 146

from_response_body() (*tensorbay.label.label_mask.RemoteInstanceMask* class method), 201

from_response_body() (*tensorbay.label.label_mask.RemotePanopticMask* class method), 202

from_response_body() (*tensorbay.label.label_mask.RemoteSemanticMask* class method), 200

from_xywh() (*tensorbay.geometry.box.Box2D* class method), 148

from_xywh() (*tensorbay.label.label_box.LabeledBox2D* class method), 182

FSDD() (*in module tensorbay.opendataset.FSDD.loader*), 269

FusionDataset (*class* in *tensorbay.dataset.dataset*), 144

FusionDatasetClient (*class* in *tensorbay.client.dataset*), 116

FusionSegment (*class* in *tensorbay.dataset.segment*), 145

FusionSegmentClient (*class* in *tensorbay.client.segment*), 130

fx (*tensorbay.sensor.intrinsics.CameraMatrix* attribute), 230

fy (*tensorbay.sensor.intrinsics.CameraMatrix* attribute), 230

G

GAS (*class* in *tensorbay.client.gas*), 118

get() (*tensorbay.client.lazy.LazyItem* method), 121

get() (*tensorbay.utility.user.UserMapping* method), 254

get_auth_storage_config() (*tensorbay.client.gas.GAS* method), 118

get_branch() (*tensorbay.client.version.VersionControlClient* method), 139

get_callback_body() (*tensorbay.dataset.data.Data* method), 140

get_callback_body() (*tensorbay.label.label_mask.InstanceMask* method), 199

get_callback_body() (*tensorbay.label.label_mask.PanopticMask* method), 200

get_callback_body() (*tensorbay.label.label_mask.SemanticMask* method), 198

get_catalog() (*tensorbay.client.dataset.DatasetClientBase* method), 114

get_category_to_index() (*tensorbay.label.supports.CategoriesMixin* method), 228

get_category_to_index() (*tensorbay.label.supports.MaskCategoriesMixin* method), 229

get_cloud_client() (*tensorbay.client.gas.GAS* method), 118

get_commit() (*tensorbay.client.version.VersionControlClient* method), 138

get_dataset() (*tensorbay.client.gas.GAS* method), 119

get_draft() (*tensorbay.client.version.VersionControlClient* method), 137

get_index_to_category() (*tensorbay.label.supports.CategoriesMixin* method), 228

get_index_to_category() (*tensorbay.label.supports.MaskCategoriesMixin* method), 229

- `get_notes()` (*tensorbay.client.dataset.DatasetClientBase* method), 114
- `get_or_create_segment()` (*tensorbay.client.dataset.DatasetClient* method), 114
- `get_or_create_segment()` (*tensorbay.client.dataset.FusionDatasetClient* method), 116
- `get_segment()` (*tensorbay.client.dataset.DatasetClient* method), 115
- `get_segment()` (*tensorbay.client.dataset.FusionDatasetClient* method), 117
- `get_sensors()` (*tensorbay.client.segment.FusionSegmentClient* method), 130
- `get_status_info()` (*tensorbay.client.status.Status* method), 131
- `get_tag()` (*tensorbay.client.version.VersionControlClient* method), 139
- `get_user()` (*tensorbay.client.gas.GAS* method), 118
- `glob()` (in module *tensorbay.opendataset._utility*), 283
- ## H
- `HalpeFullBody()` (in module *tensorbay.opendataset.HalpeFullBody.loader*), 270
- `HardHatWorkers()` (in module *tensorbay.opendataset.HardHatWorkers.loader*), 270
- `HeadPoseImage()` (in module *tensorbay.opendataset.HeadPoseImage.loader*), 271
- `height` (*tensorbay.geometry.box.Box2D* property), 150
- ## I
- `ImageEmotionAbstract()` (in module *tensorbay.opendataset.ImageEmotion.loader*), 271
- `ImageEmotionArtphoto()` (in module *tensorbay.opendataset.ImageEmotion.loader*), 271
- `import_auth_data()` (*tensorbay.client.segment.SegmentClient* method), 129
- `index()` (*tensorbay.client.lazy.PagingList* method), 123
- `index()` (*tensorbay.utility.user.UserSequence* method), 253
- `InitPage` (class in *tensorbay.client.lazy*), 122
- `insert()` (*tensorbay.client.lazy.PagingList* method), 122
- `insert()` (*tensorbay.utility.user.UserMutableSequence* method), 253
- `instance` (*tensorbay.label.label_box.LabeledBox2D* attribute), 181
- `instance` (*tensorbay.label.label_box.LabeledBox3D* attribute), 185
- `instance` (*tensorbay.label.label_keypoints.LabeledKeypoints2D* attribute), 193
- `instance` (*tensorbay.label.label_polygon.LabeledMultiPolygon* attribute), 208
- `instance` (*tensorbay.label.label_polygon.LabeledPolygon* attribute), 207
- `instance` (*tensorbay.label.label_polygon.LabeledRLE* attribute), 210
- `instance` (*tensorbay.label.label_polyline.LabeledMultiPolyline2D* attribute), 217
- `instance` (*tensorbay.label.label_polyline.LabeledPolyline2D* attribute), 214
- `InstanceMask` (class in *tensorbay.label.label_mask*), 199
- `InstanceMaskBase` (class in *tensorbay.label.label_mask*), 198
- `InstanceMaskSubcatalog` (class in *tensorbay.label.label_mask*), 195
- `intrinsics` (*tensorbay.sensor.sensor.Camera* attribute), 243
- `InvalidParamsError`, 111, 257
- `inverse()` (*tensorbay.geometry.transform.Transform3D* method), 166
- `iou()` (*tensorbay.geometry.box.Box2D* static method), 147
- `iou()` (*tensorbay.geometry.box.Box3D* class method), 152
- `is_draft` (*tensorbay.client.status.Status* property), 131
- `is_internal` (*tensorbay.client.requests.Config* attribute), 125
- `is_sample` (*tensorbay.label.label_sentence.SentenceSubcatalog* attribute), 219
- `is_tracking` (*tensorbay.label.label_box.Box2DSubcatalog* attribute), 180
- `is_tracking` (*tensorbay.label.label_box.Box3DSubcatalog* attribute), 184
- `is_tracking` (*tensorbay.label.label_keypoints.Keypoints2DSubcatalog* attribute), 190
- `is_tracking` (*tensorbay.label.label_mask.InstanceMaskSubcatalog* attribute), 196
- `is_tracking` (*tensorbay.label.label_mask.PanopticMaskSubcatalog* attribute), 197
- `is_tracking` (*tensorbay.label.label_mask.SemanticMaskSubcatalog* attribute), 195
- `is_tracking` (*tensorbay.label.label_polygon.MultiPolygonSubcatalog* attribute), 204
- `is_tracking` (*tensorbay.label.label_polygon.PolygonSubcatalog* attribute), 203
- `is_tracking` (*tensorbay.label.label_polygon.RLESubcatalog* attribute), 205
- `is_tracking` (*tensorbay.label.label_polyline.MultiPolyline2DSubcatalog* attribute), 215

- `is_tracking` (*tensorbay.label.label_polyline.Polyline2DSubcatalog* attribute), 212
- `is_tracking` (*tensorbay.label.supports.IsTrackingMixin* attribute), 228
- `IsTrackingMixin` (class in *tensorbay.label.supports*), 228
- `Items` (class in *tensorbay.label.attributes*), 170
- `items` (*tensorbay.client.lazy.InitPage* attribute), 122
- `items` (*tensorbay.client.lazy.LazyPage* attribute), 122
- `items` (*tensorbay.label.attributes.AttributeInfo* attribute), 173
- `items` (*tensorbay.label.attributes.Items* attribute), 171
- `items()` (*tensorbay.utility.user.UserMapping* method), 254
- ## J
- `JHU_CROWD()` (in module *tensorbay.opendataset.JHU_CROWD.loader*), 272
- ## K
- `KenyanFoodOrNonfood()` (in module *tensorbay.opendataset.KenyanFood.loader*), 272
- `KenyanFoodType()` (in module *tensorbay.opendataset.KenyanFood.loader*), 273
- `Keypoint2D` (class in *tensorbay.geometry.keypoint*), 154
- `keypoints` (*tensorbay.label.label_keypoints.Keypoints2DSubcatalog* property), 191
- `Keypoints2D` (class in *tensorbay.geometry.keypoint*), 155
- `Keypoints2DSubcatalog` (class in *tensorbay.label.label_keypoints*), 189
- `KeypointsInfo` (class in *tensorbay.label.supports*), 226
- `keys()` (*tensorbay.dataset.dataset.DatasetBase* method), 144
- `keys()` (*tensorbay.dataset.dataset.Notes* method), 143
- `keys()` (*tensorbay.utility.name.NameList* method), 251
- `keys()` (*tensorbay.utility.name.SortedNameList* method), 252
- `keys()` (*tensorbay.utility.user.UserMapping* method), 254
- `KylbergTexture()` (in module *tensorbay.opendataset.KylbergTexture.loader*), 274
- ## L
- `Label` (class in *tensorbay.label.label*), 178
- `label` (*tensorbay.dataset.data.AuthData* attribute), 142
- `label` (*tensorbay.dataset.data.Data* attribute), 140
- `label` (*tensorbay.dataset.data.DataBase* attribute), 140
- `label` (*tensorbay.dataset.data.RemoteData* attribute), 141
- `LabeledBox2D` (class in *tensorbay.label.label_box*), 181
- `LabeledBox3D` (class in *tensorbay.label.label_box*), 185
- `LabeledKeypoints2D` (class in *tensorbay.label.label_keypoints*), 192
- `LabeledMultiPolygon` (class in *tensorbay.label.label_polygon*), 208
- `LabeledMultiPolyline2D` (class in *tensorbay.label.label_polyline*), 216
- `LabeledPolygon` (class in *tensorbay.label.label_polygon*), 206
- `LabeledPolyline2D` (class in *tensorbay.label.label_polyline*), 213
- `LabeledRLE` (class in *tensorbay.label.label_polygon*), 210
- `LabeledSentence` (class in *tensorbay.label.label_sentence*), 221
- `last_callback()` (*tensorbay.client.requests.MultiCallbackTask* method), 128
- `LazyItem` (class in *tensorbay.client.lazy*), 121
- `LazyPage` (class in *tensorbay.client.lazy*), 121
- `LeedsSportsPose()` (in module *tensorbay.opendataset.LeedsSportsPose.loader*), 276
- `lexicon` (*tensorbay.label.label_sentence.SentenceSubcatalog* attribute), 219
- `Lidar` (class in *tensorbay.sensor.sensor*), 242
- `LISATrafficLight()` (in module *tensorbay.opendataset.LISATrafficLight.loader*), 274
- `LISATrafficSign()` (in module *tensorbay.opendataset.LISATrafficSign.loader*), 275
- `list_auth_data()` (*tensorbay.client.cloud_storage.CloudClient* method), 113
- `list_auth_storage_configs()` (*tensorbay.client.gas.GAS* method), 118
- `list_branches()` (*tensorbay.client.version.VersionControlClient* method), 139
- `list_commits()` (*tensorbay.client.version.VersionControlClient* method), 138
- `list_data()` (*tensorbay.client.segment.SegmentClient* method), 130
- `list_data_paths()` (*tensorbay.client.segment.SegmentClient* method), 130
- `list_dataset_names()` (*tensorbay.client.gas.GAS* method), 119
- `list_drafts()` (*tensorbay.client.version.VersionControlClient* method), 138
- `list_frames()` (*tensorbay.client.segment.FusionSegmentClient* method), 130

`method`), 131
`list_mask_urls()` (`tensorbay.client.segment.SegmentClient` `method`), 130
`list_segment_names()` (`tensorbay.client.dataset.DatasetClientBase` `method`), 114
`list_tags()` (`tensorbay.client.version.VersionControlClient` `method`), 139
`list_urls()` (`tensorbay.client.segment.FusionSegmentClient` `method`), 131
`list_urls()` (`tensorbay.client.segment.SegmentClient` `method`), 130
`load_catalog()` (`tensorbay.dataset.dataset.DatasetBase` `method`), 144
`loads()` (`tensorbay.client.struct.Commit` `class method`), 134
`loads()` (`tensorbay.client.struct.Draft` `class method`), 136
`loads()` (`tensorbay.client.struct.TeamInfo` `class method`), 132
`loads()` (`tensorbay.client.struct.User` `class method`), 134
`loads()` (`tensorbay.client.struct.UserInfo` `class method`), 133
`loads()` (`tensorbay.dataset.dataset.Notes` `class method`), 143
`loads()` (`tensorbay.geometry.box.Box2D` `class method`), 148
`loads()` (`tensorbay.geometry.box.Box3D` `class method`), 151
`loads()` (`tensorbay.geometry.keypoint.Keypoint2D` `class method`), 154
`loads()` (`tensorbay.geometry.keypoint.Keypoints2D` `class method`), 155
`loads()` (`tensorbay.geometry.point_list.MultiPointList2D` `class method`), 157
`loads()` (`tensorbay.geometry.point_list.PointList2D` `class method`), 156
`loads()` (`tensorbay.geometry.polygon.MultiPolygon` `class method`), 158
`loads()` (`tensorbay.geometry.polygon.Polygon` `class method`), 157
`loads()` (`tensorbay.geometry.polygon.RLE` `class method`), 159
`loads()` (`tensorbay.geometry.polyline.MultiPolyline2D` `class method`), 162
`loads()` (`tensorbay.geometry.polyline.Polyline2D` `class method`), 161
`loads()` (`tensorbay.geometry.transform.Transform3D` `class method`), 163
`loads()` (`tensorbay.geometry.vector.Vector` `static method`), 167
`loads()` (`tensorbay.geometry.vector.Vector2D` `class method`), 167
`loads()` (`tensorbay.geometry.vector.Vector3D` `class method`), 169
`loads()` (`tensorbay.label.attributes.AttributeInfo` `class method`), 174
`loads()` (`tensorbay.label.attributes.Items` `class method`), 171
`loads()` (`tensorbay.label.basic.SubcatalogBase` `class method`), 175
`loads()` (`tensorbay.label.catalog.Catalog` `class method`), 176
`loads()` (`tensorbay.label.label.Label` `class method`), 178
`loads()` (`tensorbay.label.label_box.LabeledBox2D` `class method`), 182
`loads()` (`tensorbay.label.label_box.LabeledBox3D` `class method`), 186
`loads()` (`tensorbay.label.label_classification.Classification` `class method`), 189
`loads()` (`tensorbay.label.label_keypoints.LabeledKeypoints2D` `class method`), 193
`loads()` (`tensorbay.label.label_polygon.LabeledMultiPolygon` `class method`), 209
`loads()` (`tensorbay.label.label_polygon.LabeledPolygon` `class method`), 207
`loads()` (`tensorbay.label.label_polygon.LabeledRLE` `class method`), 211
`loads()` (`tensorbay.label.label_polyline.LabeledMultiPolyline2D` `class method`), 217
`loads()` (`tensorbay.label.label_polyline.LabeledPolyline2D` `class method`), 214
`loads()` (`tensorbay.label.label_sentence.LabeledSentence` `class method`), 223
`loads()` (`tensorbay.label.label_sentence.Word` `class method`), 221
`loads()` (`tensorbay.label.supports.CategoryInfo` `class method`), 225
`loads()` (`tensorbay.label.supports.KeypointsInfo` `class method`), 227
`loads()` (`tensorbay.sensor.intrinsics.CameraIntrinsics` `class method`), 236
`loads()` (`tensorbay.sensor.intrinsics.CameraMatrix` `class method`), 231
`loads()` (`tensorbay.sensor.intrinsics.DistortionCoefficients` `class method`), 233
`loads()` (`tensorbay.sensor.sensor.Camera` `class method`), 244
`loads()` (`tensorbay.sensor.sensor.Sensor` `static method`), 240
`loads()` (`tensorbay.sensor.sensor.Sensors` `class method`), 247
`locked()` (in module `tensorbay.utility.common`), 251

M

`MaskCategoriesMixin` (class in `tensor-`

- bay.label.supports*), 229
- MaskCategoryInfo (class in *tensorbay.label.supports*), 225
- max_retries (*tensorbay.client.requests.Config* attribute), 125
- maximum (*tensorbay.label.attributes.AttributeInfo* attribute), 173
- maximum (*tensorbay.label.attributes.Items* attribute), 171
- minimum (*tensorbay.label.attributes.AttributeInfo* attribute), 173
- minimum (*tensorbay.label.attributes.Items* attribute), 171
- module
 - tensorbay.client.cloud_storage*, 112
 - tensorbay.client.dataset*, 113
 - tensorbay.client.gas*, 118
 - tensorbay.client.lazy*, 121
 - tensorbay.client.log*, 123
 - tensorbay.client.requests*, 125
 - tensorbay.client.segment*, 128
 - tensorbay.client.status*, 131
 - tensorbay.client.struct*, 132
 - tensorbay.client.version*, 136
 - tensorbay.dataset.data*, 140
 - tensorbay.dataset.dataset*, 142
 - tensorbay.dataset.frame*, 146
 - tensorbay.dataset.segment*, 144
 - tensorbay.exception*, 255
 - tensorbay.geometry.box*, 147
 - tensorbay.geometry.keypoint*, 154
 - tensorbay.geometry.point_list*, 156
 - tensorbay.geometry.polygon*, 157
 - tensorbay.geometry.polyline*, 160
 - tensorbay.geometry.transform*, 163
 - tensorbay.geometry.vector*, 166
 - tensorbay.label.attributes*, 170
 - tensorbay.label.basic*, 175
 - tensorbay.label.catalog*, 176
 - tensorbay.label.label*, 178
 - tensorbay.label.label_box*, 179
 - tensorbay.label.label_classification*, 187
 - tensorbay.label.label_keypoints*, 189
 - tensorbay.label.label_mask*, 194
 - tensorbay.label.label_polygon*, 202
 - tensorbay.label.label_polyline*, 212
 - tensorbay.label.label_sentence*, 218
 - tensorbay.label.supports*, 224
 - tensorbay.opendataset._utility*, 282
 - tensorbay.opendataset.AADB*.loader, 259
 - tensorbay.opendataset.AnimalPose*.loader, 259
 - tensorbay.opendataset.AnimalsWithAttributes2*.loader, 260
 - tensorbay.opendataset.BDD100K*.loader, 260
 - tensorbay.opendataset.BioIDFace*.loader, 262
 - tensorbay.opendataset.BSTLD*.loader, 261
 - tensorbay.opendataset.CACD*.loader, 262
 - tensorbay.opendataset.CADC*.loader, 262
 - tensorbay.opendataset.CarConnection*.loader, 266
 - tensorbay.opendataset.CCPD*.loader, 263
 - tensorbay.opendataset.CoinImage*.loader, 266
 - tensorbay.opendataset.CompCars*.loader, 266
 - tensorbay.opendataset.COVID_CT*.loader, 265
 - tensorbay.opendataset.COVIDChestXRay*.loader, 264
 - tensorbay.opendataset.DeepRoute*.loader, 267
 - tensorbay.opendataset.DogsVsCats*.loader, 267
 - tensorbay.opendataset.DownsamplingImagenet*.loader, 268
 - tensorbay.opendataset.Elpy*.loader, 268
 - tensorbay.opendataset.FLIC*.loader, 269
 - tensorbay.opendataset.Flower*.loader, 269
 - tensorbay.opendataset.FSDD*.loader, 269
 - tensorbay.opendataset.HalpeFullBody*.loader, 270
 - tensorbay.opendataset.HardHatWorkers*.loader, 270
 - tensorbay.opendataset.HeadPoseImage*.loader, 271
 - tensorbay.opendataset.ImageEmotion*.loader, 271
 - tensorbay.opendataset.JHU_CROWD*.loader, 272
 - tensorbay.opendataset.KenyanFood*.loader, 272
 - tensorbay.opendataset.KylbergTexture*.loader, 274
 - tensorbay.opendataset.LeedsSportsPose*.loader, 276
 - tensorbay.opendataset.LISATrafficLight*.loader, 274
 - tensorbay.opendataset.LISATrafficSign*.loader, 275
 - tensorbay.opendataset.NeolixOD*.loader, 277
 - tensorbay.opendataset.Newsgroups20*.loader, 277
 - tensorbay.opendataset.NightOwls*.loader, 278
 - tensorbay.opendataset.nuScenes*.loader, 283

- tensorbay.opendataset.RP2K.loader, 278
 - tensorbay.opendataset.THCHS30.loader, 279
 - tensorbay.opendataset.THUCNews.loader, 279
 - tensorbay.opendataset.TLR.loader, 280
 - tensorbay.opendataset.UAVDT.loader, 280
 - tensorbay.opendataset.VOC2012ActionClassification.loader, 281
 - tensorbay.opendataset.VOC2012Detection.loader, 281
 - tensorbay.opendataset.WIDER_FACE.loader, 282
 - tensorbay.sensor.intrinsics, 230
 - tensorbay.sensor.sensor, 239
 - tensorbay.utility.attr, 249
 - tensorbay.utility.common, 250
 - tensorbay.utility.name, 251
 - tensorbay.utility.repr, 252
 - tensorbay.utility.type, 252
 - tensorbay.utility.user, 253
 - ModuleImportError, 258
 - move_data() (tensorbay.client.segment.SegmentClient method), 129
 - move_segment() (tensorbay.client.dataset.DatasetClient method), 115
 - move_segment() (tensorbay.client.dataset.FusionDatasetClient method), 116
 - MultiCallbackTask (class in tensorbay.client.requests), 127
 - MultiPointList2D (class in tensorbay.geometry.point_list), 156
 - MultiPolygon (class in tensorbay.geometry.polygon), 158
 - MultiPolygonSubcatalog (class in tensorbay.label.label_polygon), 204
 - MultiPolyline2D (class in tensorbay.geometry.polyline), 161
 - MultiPolyline2DSubcatalog (class in tensorbay.label.label_polyline), 215
 - multithread_upload() (in module tensorbay.client.requests), 127
- ## N
- name (tensorbay.client.dataset.DatasetClientBase attribute), 113
 - name (tensorbay.client.segment.SegmentClientBase attribute), 128
 - name (tensorbay.label.supports.CategoryInfo attribute), 224
 - name (tensorbay.label.supports.MaskCategoryInfo attribute), 225
 - name (tensorbay.utility.name.NameMixin attribute), 251
 - NameConflictError, 112, 257
 - NameList (class in tensorbay.utility.name), 251
 - NameMixin (class in tensorbay.utility.name), 251
 - names (tensorbay.label.supports.KeypointsInfo attribute), 226
 - NeolixOD() (in module tensorbay.opendataset.NeolixOD.loader), 277
 - Newsgroups20() (in module tensorbay.opendataset.Newsgroups20.loader), 277
 - NightOwls() (in module tensorbay.opendataset.NightOwls.loader), 278
 - NoFileError, 112, 258
 - Notes (class in tensorbay.dataset.dataset), 142
 - notes (tensorbay.dataset.dataset.DatasetBase attribute), 143
 - number (tensorbay.label.supports.KeypointsInfo attribute), 226
 - nuScenes() (in module tensorbay.opendataset.nuScenes.loader), 283
- ## O
- open_api_do() (tensorbay.client.requests.Client method), 126
 - OpenDatasetError, 112, 258
 - OperationError, 256
- ## P
- page (tensorbay.client.lazy.LazyItem attribute), 121
 - PagingList (class in tensorbay.client.lazy), 122
 - PanopticMask (class in tensorbay.label.label_mask), 199
 - PanopticMaskBase (class in tensorbay.label.label_mask), 198
 - PanopticMaskSubcatalog (class in tensorbay.label.label_mask), 196
 - parent_categories (tensorbay.label.attributes.AttributeInfo attribute), 173
 - parent_categories (tensorbay.label.supports.KeypointsInfo attribute), 226
 - path (tensorbay.dataset.data.AuthData attribute), 142
 - path (tensorbay.dataset.data.Data attribute), 140
 - path (tensorbay.dataset.data.RemoteData attribute), 141
 - phone (tensorbay.label.label_sentence.LabeledSentence attribute), 222
 - PointList2D (class in tensorbay.geometry.point_list), 156
 - Polygon (class in tensorbay.geometry.polygon), 157
 - PolygonSubcatalog (class in tensorbay.label.label_polygon), 202
 - Polyline2D (class in tensorbay.geometry.polyline), 160

Polyline2DSubcatalog (class in *tensorbay.label.label_polyline*), 212
 pop() (*tensorbay.client.lazy.PagingList* method), 123
 pop() (*tensorbay.utility.user.UserMutableMapping* method), 254
 pop() (*tensorbay.utility.user.UserMutableSequence* method), 254
 popitem() (*tensorbay.utility.user.UserMutableMapping* method), 255
 project() (*tensorbay.sensor.intrinsics.CameraIntrinsics* method), 238
 project() (*tensorbay.sensor.intrinsics.CameraMatrix* method), 232
 pull() (*tensorbay.client.lazy.LazyPage* method), 122

R

Radar (class in *tensorbay.sensor.sensor*), 243
 RemoteData (class in *tensorbay.dataset.data*), 141
 RemoteInstanceMask (class in *tensorbay.label.label_mask*), 201
 RemotePanopticMask (class in *tensorbay.label.label_mask*), 202
 RemoteSemanticMask (class in *tensorbay.label.label_mask*), 200
 remove() (*tensorbay.utility.user.UserMutableSequence* method), 254
 rename_dataset() (*tensorbay.client.gas.GAS* method), 119
 ReprMixin (class in *tensorbay.utility.repr*), 252
 ReprType (class in *tensorbay.utility.repr*), 252
 request() (*tensorbay.client.requests.UserSession* method), 126
 RequestLogging (class in *tensorbay.client.log*), 123
 RequestParamsMissingError, 112, 257
 ResourceNotExistError, 112, 257
 response (*tensorbay.exception.InvalidParamsError* attribute), 257
 response (*tensorbay.exception.NameConflictError* attribute), 257
 response (*tensorbay.exception.ResponseError* attribute), 257
 ResponseError, 111, 256
 ResponseLogging (class in *tensorbay.client.log*), 123
 ResponseSystemError, 112, 258
 ReturnGenerator (class in *tensorbay.client.lazy*), 121
 reverse() (*tensorbay.client.lazy.PagingList* method), 123
 reverse() (*tensorbay.utility.user.UserMutableSequence* method), 254
 RLE (class in *tensorbay.geometry.polygon*), 159
 RLESubcatalog (class in *tensorbay.label.label_polygon*), 205
 rotation (*tensorbay.geometry.box.Box3D* property), 152

rotation (*tensorbay.geometry.transform.Transform3D* property), 164
 RP2K() (in module *tensorbay.opendataset.RP2K.loader*), 278

S

sample_rate (*tensorbay.label.label_sentence.SentenceSubcatalog* attribute), 219
 Segment (class in *tensorbay.dataset.segment*), 144
 SegmentClient (class in *tensorbay.client.segment*), 128
 SegmentClientBase (class in *tensorbay.client.segment*), 128
 SemanticMask (class in *tensorbay.label.label_mask*), 198
 SemanticMaskBase (class in *tensorbay.label.label_mask*), 198
 SemanticMaskSubcatalog (class in *tensorbay.label.label_mask*), 194
 send() (*tensorbay.client.requests.TimeoutHTTPAdapter* method), 125
 Sensor (class in *tensorbay.sensor.sensor*), 240
 Sensors (class in *tensorbay.sensor.sensor*), 247
 sensors (*tensorbay.dataset.segment.FusionSegment* property), 146
 SensorType (class in *tensorbay.sensor.sensor*), 239
 sentence (*tensorbay.label.label_sentence.LabeledSentence* attribute), 222
 SentenceSubcatalog (class in *tensorbay.label.label_sentence*), 218
 session (*tensorbay.client.requests.Client* property), 126
 set_camera_matrix() (*tensorbay.sensor.intrinsics.CameraIntrinsics* method), 237
 set_camera_matrix() (*tensorbay.sensor.sensor.Camera* method), 245
 set_distortion_coefficients() (*tensorbay.sensor.intrinsics.CameraIntrinsics* method), 238
 set_distortion_coefficients() (*tensorbay.sensor.sensor.Camera* method), 246
 set_extrinsics() (*tensorbay.sensor.sensor.Sensor* method), 241
 set_rotation() (*tensorbay.geometry.transform.Transform3D* method), 165
 set_rotation() (*tensorbay.sensor.sensor.Sensor* method), 242
 set_translation() (*tensorbay.geometry.transform.Transform3D* method), 164
 set_translation() (*tensorbay.sensor.sensor.Sensor* method), 241
 setdefault() (*tensorbay.utility.user.UserMutableMapping* method),

- 255
- `similarity()` (*tensorbay.geometry.polyline.Polyline2D* static method), 161
- `size` (*tensorbay.geometry.box.Box3D* property), 153
- `size` (*tensorbay.label.label_box.LabeledBox3D* attribute), 185
- `skeleton` (*tensorbay.label.supports.KeypointsInfo* attribute), 226
- `skew` (*tensorbay.sensor.intrinsics.CameraMatrix* attribute), 231
- `sort()` (*tensorbay.dataset.segment.Segment* method), 145
- `SortedNameList` (class in *tensorbay.utility.name*), 251
- `spell` (*tensorbay.label.label_sentence.LabeledSentence* attribute), 222
- `Status` (class in *tensorbay.client.status*), 131
- `status` (*tensorbay.client.dataset.DatasetClientBase* attribute), 113
- `status` (*tensorbay.client.segment.SegmentClientBase* attribute), 128
- `status` (*tensorbay.client.version.VersionControlClient* property), 137
- `StatusError`, 111, 256
- `SubcatalogBase` (class in *tensorbay.label.basic*), 175
- ## T
- `Tag` (class in *tensorbay.client.struct*), 135
- `target_remote_path` (*tensorbay.dataset.data.Data* attribute), 140
- `TBRNError`, 111, 258
- `TeamInfo` (class in *tensorbay.client.struct*), 132
- `tensorbay.client.cloud_storage`
module, 112
- `tensorbay.client.dataset`
module, 113
- `tensorbay.client.gas`
module, 118
- `tensorbay.client.lazy`
module, 121
- `tensorbay.client.log`
module, 123
- `tensorbay.client.requests`
module, 125
- `tensorbay.client.segment`
module, 128
- `tensorbay.client.status`
module, 131
- `tensorbay.client.struct`
module, 132
- `tensorbay.client.version`
module, 136
- `tensorbay.dataset.data`
module, 140
- `tensorbay.dataset.dataset`
module, 142
- `tensorbay.dataset.frame`
module, 146
- `tensorbay.dataset.segment`
module, 144
- `tensorbay.exception`
module, 255
- `tensorbay.geometry.box`
module, 147
- `tensorbay.geometry.keypoint`
module, 154
- `tensorbay.geometry.point_list`
module, 156
- `tensorbay.geometry.polygon`
module, 157
- `tensorbay.geometry.polyline`
module, 160
- `tensorbay.geometry.transform`
module, 163
- `tensorbay.geometry.vector`
module, 166
- `tensorbay.label.attributes`
module, 170
- `tensorbay.label.basic`
module, 175
- `tensorbay.label.catalog`
module, 176
- `tensorbay.label.label`
module, 178
- `tensorbay.label.label_box`
module, 179
- `tensorbay.label.label_classification`
module, 187
- `tensorbay.label.label_keypoints`
module, 189
- `tensorbay.label.label_mask`
module, 194
- `tensorbay.label.label_polygon`
module, 202
- `tensorbay.label.label_polyline`
module, 212
- `tensorbay.label.label_sentence`
module, 218
- `tensorbay.label.supports`
module, 224
- `tensorbay.opendataset._utility`
module, 282
- `tensorbay.opendataset.AADB.loader`
module, 259
- `tensorbay.opendataset.AnimalPose.loader`
module, 259
- `tensorbay.opendataset.AnimalsWithAttributes2.loader`
module, 260
- `tensorbay.opendataset.BDD100K.loader`

module, 260
tensorbay.opendataset.BioIDFace.loader
 module, 262
tensorbay.opendataset.BSTLD.loader
 module, 261
tensorbay.opendataset.CACD.loader
 module, 262
tensorbay.opendataset.CADC.loader
 module, 262
tensorbay.opendataset.CarConnection.loader
 module, 266
tensorbay.opendataset.CCPD.loader
 module, 263
tensorbay.opendataset.CoinImage.loader
 module, 266
tensorbay.opendataset.CompCars.loader
 module, 266
tensorbay.opendataset.COVID_CT.loader
 module, 265
tensorbay.opendataset.COVIDChestXRay.loader
 module, 264
tensorbay.opendataset.DeepRoute.loader
 module, 267
tensorbay.opendataset.DogsVsCats.loader
 module, 267
tensorbay.opendataset.DownsamplingImagenet.loader
 module, 268
tensorbay.opendataset.Elpv.loader
 module, 268
tensorbay.opendataset.FLIC.loader
 module, 269
tensorbay.opendataset.Flower.loader
 module, 269
tensorbay.opendataset.FSDD.loader
 module, 269
tensorbay.opendataset.HalpeFullBody.loader
 module, 270
tensorbay.opendataset.HardHatWorkers.loader
 module, 270
tensorbay.opendataset.HeadPoseImage.loader
 module, 271
tensorbay.opendataset.ImageEmotion.loader
 module, 271
tensorbay.opendataset.JHU_CROWD.loader
 module, 272
tensorbay.opendataset.KenyanFood.loader
 module, 272
tensorbay.opendataset.KylbergTexture.loader
 module, 274
tensorbay.opendataset.LeedsSportsPose.loader
 module, 276
tensorbay.opendataset.LISATrafficLight.loader
 module, 274
tensorbay.opendataset.LISATrafficSign.loader
 module, 275
tensorbay.opendataset.NeolixOD.loader
 module, 277
tensorbay.opendataset.Newsgroups20.loader
 module, 277
tensorbay.opendataset.NightOwls.loader
 module, 278
tensorbay.opendataset.nuScenes.loader
 module, 283
tensorbay.opendataset.RP2K.loader
 module, 278
tensorbay.opendataset.THCHS30.loader
 module, 279
tensorbay.opendataset.THUCNews.loader
 module, 279
tensorbay.opendataset.TLR.loader
 module, 280
tensorbay.opendataset.UAVDT.loader
 module, 280
tensorbay.opendataset.VOC2012ActionClassification.loader
 module, 281
tensorbay.opendataset.VOC2012Detection.loader
 module, 281
tensorbay.opendataset.WIDER_FACE.loader
 module, 282
tensorbay.sensor.intrinsics
 module, 230
tensorbay.sensor.sensor
 module, 239
tensorbay.utility.attr
 module, 249
tensorbay.utility.common
 module, 250
tensorbay.utility.name
 module, 251
tensorbay.utility.repr
 module, 252
tensorbay.utility.type
 module, 252
tensorbay.utility.user
 module, 253
TensorBayException, 111, 256
text (*tensorbay.label.label_sentence.Word* attribute),
 220
THCHS30() (in module *tensorbay.opendataset.THCHS30.loader*), 279
THUCNews() (in module *tensorbay.opendataset.THUCNews.loader*), 279
timeout (*tensorbay.client.requests.Config* attribute), 125
TimeoutHTTPAdapter (class in *tensorbay.client.requests*), 125
timestamp (*tensorbay.dataset.data.AuthData* attribute),
 142
timestamp (*tensorbay.dataset.data.Data* attribute), 140

- timestamp (*tensorbay.dataset.data.DataBase* attribute), 140
- timestamp (*tensorbay.dataset.data.RemoteData* attribute), 141
- tl (*tensorbay.geometry.box.Box2D* property), 149
- TLR() (in module *tensorbay.opendataset.TLR.loader*), 280
- total_count (*tensorbay.client.lazy.InitPage* attribute), 122
- Tqdm (class in *tensorbay.client.requests*), 127
- transform (*tensorbay.geometry.box.Box3D* property), 153
- transform (*tensorbay.label.label_box.LabeledBox3D* attribute), 185
- Transform3D (class in *tensorbay.geometry.transform*), 163
- translation (*tensorbay.geometry.box.Box3D* property), 152
- translation (*tensorbay.geometry.transform.Transform3D* property), 164
- type (*tensorbay.label.attributes.AttributeInfo* attribute), 173
- type (*tensorbay.label.attributes.Items* attribute), 171
- type (*tensorbay.utility.type.TypeEnum* property), 252
- TypeEnum (class in *tensorbay.utility.type*), 252
- TypeMixin (class in *tensorbay.utility.type*), 252
- TypeRegister (class in *tensorbay.utility.type*), 253
- ## U
- UAVDT() (in module *tensorbay.opendataset.UAVDT.loader*), 280
- UnauthorizedError, 112, 258
- uniform_frechet_distance() (*tensorbay.geometry.polyline.Polyline2D* static method), 160
- update() (*tensorbay.utility.user.UserMutableMapping* method), 255
- update_callback() (*tensorbay.client.requests.Tqdm* method), 127
- update_dataset() (*tensorbay.client.gas.GAS* method), 119
- update_draft() (*tensorbay.client.version.VersionControlClient* method), 138
- update_for_skip() (*tensorbay.client.requests.Tqdm* method), 127
- update_notes() (*tensorbay.client.dataset.DatasetClientBase* method), 113
- upload_catalog() (*tensorbay.client.dataset.DatasetClientBase* method), 114
- upload_data() (*tensorbay.client.segment.SegmentClient* method), 129
- upload_dataset() (*tensorbay.client.gas.GAS* method), 120
- upload_file() (*tensorbay.client.segment.SegmentClient* method), 128
- upload_frame() (*tensorbay.client.segment.FusionSegmentClient* method), 130
- upload_label() (*tensorbay.client.segment.SegmentClient* method), 129
- upload_segment() (*tensorbay.client.dataset.DatasetClient* method), 115
- upload_segment() (*tensorbay.client.dataset.FusionDatasetClient* method), 117
- upload_sensor() (*tensorbay.client.segment.FusionSegmentClient* method), 130
- upper() (in module *tensorbay.utility.attr*), 250
- User (class in *tensorbay.client.struct*), 133
- UserInfo (class in *tensorbay.client.struct*), 132
- UserMapping (class in *tensorbay.utility.user*), 254
- UserMutableMapping (class in *tensorbay.utility.user*), 254
- UserMutableSequence (class in *tensorbay.utility.user*), 253
- UserSequence (class in *tensorbay.utility.user*), 253
- UserSession (class in *tensorbay.client.requests*), 126
- UtilityError, 258
- ## V
- v (*tensorbay.geometry.keypoint.Keypoint2D* property), 155
- value (*tensorbay.client.lazy.ReturnGenerator* attribute), 121
- values() (*tensorbay.utility.user.UserMapping* method), 254
- Vector (class in *tensorbay.geometry.vector*), 166
- Vector2D (class in *tensorbay.geometry.vector*), 167
- Vector3D (class in *tensorbay.geometry.vector*), 168
- VersionControlClient (class in *tensorbay.client.version*), 136
- visible (*tensorbay.label.supports.KeypointsInfo* attribute), 226
- VOC2012ActionClassification() (in module *tensorbay.opendataset.VOC2012ActionClassification.loader*), 281
- VOC2012Detection() (in module *tensorbay.opendataset.VOC2012Detection.loader*), 281
- volume() (*tensorbay.geometry.box.Box3D* method), 153

W

`WIDER_FACE()` (in module `tensorbay.opendataset.WIDER_FACE.loader`), 282

`width` (`tensorbay.geometry.box.Box2D` property), 150

`Word` (class in `tensorbay.label.label_sentence`), 220

`work()` (`tensorbay.client.requests.MultiCallbackTask` method), 127

X

`x` (`tensorbay.geometry.vector.Vector2D` property), 168

`x` (`tensorbay.geometry.vector.Vector3D` property), 169

`xmax` (`tensorbay.geometry.box.Box2D` property), 149

`xmin` (`tensorbay.geometry.box.Box2D` property), 148

Y

`y` (`tensorbay.geometry.vector.Vector2D` property), 168

`y` (`tensorbay.geometry.vector.Vector3D` property), 169

`ymax` (`tensorbay.geometry.box.Box2D` property), 149

`ymin` (`tensorbay.geometry.box.Box2D` property), 148

Z

`z` (`tensorbay.geometry.vector.Vector3D` property), 169