kwcoco

Release 0.2.9

Jon Crall

Aug 12, 2021

CONTENTS

If you are new, please see our getting started document: getting_started The Kitware COCO module defines a variant of the Microsoft COCO format, originally developed for the "collected images in context" object detection challenge. We are backwards compatible with the original module, but we also have improved implementations in several places, including segmentations, keypoints, annotation tracks, multi-spectral images, and videos (which represents a generic sequence of images).

A kwcoco file is a "manifest" that serves as a single reference that points to all images, categories, and annotations in a computer vision dataset. Thus, when applying an algorithm to a dataset, it is sufficient to have the algorithm take one dataset parameter: the path to the kwcoco file. Generally a kwcoco file will live in a "bundle" directory along with the data that it references, and paths in the kwcoco file will be relative to the location of the kwcoco file itself.

The main data structure in this model is largely based on the implementation in https://github.com/cocodataset/cocoapi It uses the same efficient core indexing data structures, but in our implementation the indexing can be optionally turned off, functions are silent by default (with the exception of long running processes, which optionally show progress by default). We support helper functions that add and remove images, categories, and annotations.

The *kwcoco.CocoDataset* class is capable of dynamic addition and removal of categories, images, and annotations. Has better support for keypoints and segmentation formats than the original COCO format. Despite being written in Python, this data structure is reasonably efficient.

```
>>> import kwcoco
>>> import json
>>> # Create demo data
>>> demo = CocoDataset.demo()
>>> # could also use demo.dump / demo.dumps, but this is more explicit
>>> text = json.dumps(demo.dataset)
>>> with open('demo.json', 'w') as file:
>>>     file.write(text)

>>> # Read from disk
>>> self = CocoDataset('demo.json')

>>> # Add data
>>> cid = self.add_category('Cat')
>>> gid = self.add_image('new-img.jpg')
>>> aid = self.add_annotation(image_id=gid, category_id=cid, bbox=[0, 0, 100, 100])

>>> # Remove data
>>> self.remove_annotations([aid])
>>> self.remove_images([gid])
>>> self.remove_categories([cid])

>>> # Look at data
>>> print(ub.repr2(self.basic_stats(), nl=1))
>>> print(ub.repr2(self.extended_stats(), nl=2))
>>> print(ub.repr2(self.boxsize_stats(), nl=3))
>>> print(ub.repr2(self.category_annotation_frequency()))


>>> # Inspect data
>>> import kwplot
>>> kwplot.autompl()
>>> self.show_image(gid=1)

>>> # Access single-item data via imgs, cats, anns
```

(continues on next page)

(continued from previous page)

```
>>> cid = 1
>>> self.cats[cid]
{'id': 1, 'name': 'astronaut', 'supercategory': 'human'}

>>> gid = 1
>>> self.imgs[gid]
{'id': 1, 'file_name': 'astro.png', 'url': 'https://i.imgur.com/KXhKM72.png'}

>>> aid = 3
>>> self.anns[aid]
{'id': 3, 'image_id': 1, 'category_id': 3, 'line': [326, 369, 500, 500]}

>>> # Access multi-item data via the annots and images helper objects
>>> aids = self.index.gid_to_aids[2]
>>> annots = self.annots(aids)

>>> print('annots = {}'.format(ub.repr2(annots, nl=1, sv=1)))
annots = <Annots(num=2)>

>>> annots.lookup('category_id')
[6, 4]

>>> annots.lookup('bbox')
[[37, 6, 230, 240], [124, 96, 45, 18]]

>>> # built in conversions to efficient kwimage array DataStructures
>>> print(ub.repr2(annots.detections.data))
{
    'boxes': <Boxes(xywh,
                array([[ 37.,   6., 230., 240.],
                       [124.,  96.,  45.,  18.]], dtype=float32))>,
    'class_idxs': np.array([5, 3], dtype=np.int64),
    'keypoints': <PointsList(n=2) at 0x7f07eda33220>,
    'segmentations': <PolygonList(n=2) at 0x7f086365aa60>,
}

>>> gids = list(self.imgs.keys())
>>> images = self.images(gids)
>>> print('images = {}'.format(ub.repr2(images, nl=1, sv=1)))
images = <Images(num=3)>

>>> images.lookup('file_name')
['astro.png', 'carl.png', 'stars.png']

>>> print('images.annots = {}'.format(images.annots))
images.annots = <AnnotGroups(n=3, m=3.7, s=3.9)>

>>> print('images.annots.cids = {!r}'.format(images.annots.cids))
images.annots.cids = [[1, 2, 3, 4, 5, 5, 5, 5, 5], [6, 4], []]
```

CHAPTER

ONE

COCODATASET API

The following is a logical grouping of the public kwcoco.CocoDataset API attributes and methods. See the in-code documentation for further details.

1.1 CocoDataset classmethods (via MixinCocoExtras)

- *kwcoco.CocoDataset.coerce* - Attempt to transform the input into the intended CocoDataset.
- *kwcoco.CocoDataset.demo* - Create a toy coco dataset for testing and demo puposes
- *kwcoco.CocoDataset.random* - Creates a random CocoDataset according to distribution parameters

1.2 CocoDataset classmethods (via CocoDataset)

- *kwcoco.CocoDataset.from_coco_paths* - Constructor from multiple coco file paths.
- *kwcoco.CocoDataset.from_data* - Constructor from a json dictionary
- *kwcoco.CocoDataset.from_image_paths* - Constructor from a list of images paths.

1.3 CocoDataset slots

- kwcoco.CocoDataset.index -
- kwcoco.CocoDataset.hashid -
- kwcoco.CocoDataset.hashid_parts -
- kwcoco.CocoDataset.tag -
- kwcoco.CocoDataset.dataset -
- kwcoco.CocoDataset.bundle_dpath -
- kwcoco.CocoDataset.assets_dpath -
- kwcoco.CocoDataset.cache_dpath -

1.4 CocoDataset properties

- kwcoco.CocoDataset.anns -
- kwcoco.CocoDataset.cats -
- kwcoco.CocoDataset.cid_to_aids -
- kwcoco.CocoDataset.data_fpath -
- kwcoco.CocoDataset.data_root -
- *kwcoco.CocoDataset.fpath* -
- kwcoco.CocoDataset.gid_to_aids -
- kwcoco.CocoDataset.img_root -
- kwcoco.CocoDataset.imgs -
- kwcoco.CocoDataset.n_annots -
- kwcoco.CocoDataset.n_cats -
- kwcoco.CocoDataset.n_images -
- kwcoco.CocoDataset.n_videos -
- kwcoco.CocoDataset.name_to_cat -

1.5 CocoDataset methods (via MixinCocoAddRemove)

- *kwcoco.CocoDataset.add_annotation* - Add an annotation to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.add_annotations* - Faster less-safe multi-item alternative to add_annotation.
- *kwcoco.CocoDataset.add_category* - Adds a category
- *kwcoco.CocoDataset.add_image* - Add an image to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.add_images* - Faster less-safe multi-item alternative
- *kwcoco.CocoDataset.add_video* - Add a video to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.clear_annotations* - Removes all annotations (but not images and categories)
- *kwcoco.CocoDataset.clear_images* - Removes all images and annotations (but not categories)
- *kwcoco.CocoDataset.ensure_category* - Like `add_category()`, but returns the existing category id if it already exists instead of failing. In this case all metadata is ignored.
- *kwcoco.CocoDataset.ensure_image* - Like `add_image()`,, but returns the existing image id if it already exists instead of failing. In this case all metadata is ignored.
- *kwcoco.CocoDataset.remove_annotation* - Remove a single annotation from the dataset
- *kwcoco.CocoDataset.remove_annotation_keypoints* - Removes all keypoints with a particular category
- *kwcoco.CocoDataset.remove_annotations* - Remove multiple annotations from the dataset.
- *kwcoco.CocoDataset.remove_categories* - Remove categories and all annotations in those categories. Currently does not change any hierarchy information
- *kwcoco.CocoDataset.remove_images* - Remove images and any annotations contained by them

- *kwcoco.CocoDataset.remove_keypoint_categories* - Removes all keypoints of a particular category as well as all annotation keypoints with those ids.
- *kwcoco.CocoDataset.remove_videos* - Remove videos and any images / annotations contained by them
- *kwcoco.CocoDataset.set_annotation_category* - Sets the category of a single annotation

1.6 CocoDataset methods (via MixinCocoObjects)

- *kwcoco.CocoDataset.annots* - Return vectorized annotation objects
- *kwcoco.CocoDataset.categories* - Return vectorized category objects
- *kwcoco.CocoDataset.images* - Return vectorized image objects
- *kwcoco.CocoDataset.videos* - Return vectorized video objects

1.7 CocoDataset methods (via MixinCocoStats)

- *kwcoco.CocoDataset.basic_stats* - Reports number of images, annotations, and categories.
- *kwcoco.CocoDataset.boxsize_stats* - Compute statistics about bounding box sizes.
- *kwcoco.CocoDataset.category_annotation_frequency* - Reports the number of annotations of each category
- *kwcoco.CocoDataset.category_annotation_type_frequency* - Reports the number of annotations of each type for each category
- *kwcoco.CocoDataset.conform* - Make the COCO file conform a stricter spec, infers attibutes where possible.
- *kwcoco.CocoDataset.extended_stats* - Reports number of images, annotations, and categories.
- *kwcoco.CocoDataset.find_representative_images* - Find images that have a wide array of categories. Attempt to find the fewest images that cover all categories using images that contain both a large and small number of annotations.
- *kwcoco.CocoDataset.keypoint_annotation_frequency* -
- *kwcoco.CocoDataset.stats* - This function corresponds to **:module:`kwcoco.cli.coco_stats`**.
- *kwcoco.CocoDataset.validate* - Performs checks on this coco dataset.

1.8 CocoDataset methods (via MixinCocoAccessors)

- *kwcoco.CocoDataset.category_graph* - Construct a networkx category hierarchy
- *kwcoco.CocoDataset.delayed_load* - Experimental method
- *kwcoco.CocoDataset.get_auxiliary_fpath* - Returns the full path to auxiliary data for an image
- *kwcoco.CocoDataset.get_image_fpath* - Returns the full path to the image
- *kwcoco.CocoDataset.keypoint_categories* - Construct a consistent CategoryTree representation of keypoint classes
- *kwcoco.CocoDataset.load_annot_sample* - Reads the chip of an annotation. Note this is much less efficient than using a sampler, but it doesn't require disk cache.

- *kwcoco.CocoDataset.load_image* - Reads an image from disk and
- *kwcoco.CocoDataset.object_categories* - Construct a consistent CategoryTree representation of object classes

1.9 CocoDataset methods (via CocoDataset)

- *kwcoco.CocoDataset.copy* - Deep copies this object
- *kwcoco.CocoDataset.dump* - Writes the dataset out to the json format
- *kwcoco.CocoDataset.dumps* - Writes the dataset out to the json format
- *kwcoco.CocoDataset.subset* - Return a subset of the larger coco dataset by specifying which images to port. All annotations in those images will be taken.
- *kwcoco.CocoDataset.union* - Merges multiple `CocoDataset` items into one. Names and associations are retained, but ids may be different.
- *kwcoco.CocoDataset.view_sql* - Create a cached SQL interface to this dataset suitable for large scale multiprocessing use cases.

1.10 CocoDataset methods (via MixinCocoExtras)

- *kwcoco.CocoDataset.corrupted_images* - Check for images that don't exist or can't be opened
- *kwcoco.CocoDataset.missing_images* - Check for images that don't exist
- *kwcoco.CocoDataset.rename_categories* - Rename categories with a potentially coarser categorization.
- *kwcoco.CocoDataset.reroot* - Rebase image/data paths onto a new image/data root.

1.11 CocoDataset methods (via MixinCocoDraw)

- *kwcoco.CocoDataset.draw_image* - Use kwimage to draw all annotations on an image and return the pixels as a numpy array.
- *kwcoco.CocoDataset.imread* - Loads a particular image
- *kwcoco.CocoDataset.show_image* - Use matplotlib to show an image with annotations overlaid

CHAPTER

TWO

KWCOCO

The Kitware COCO module defines a variant of the Microsoft COCO format, originally developed for the "collected images in context" object detection challenge. We are backwards compatible with the original module, but we also have improved implementations in several places, including segmentations, keypoints, annotation tracks, multi-spectral images, and videos (which represents a generic sequence of images).

A kwcoco file is a "manifest" that serves as a single reference that points to all images, categories, and annotations in a computer vision dataset. Thus, when applying an algorithm to a dataset, it is sufficient to have the algorithm take one dataset parameter: the path to the kwcoco file. Generally a kwcoco file will live in a "bundle" directory along with the data that it references, and paths in the kwcoco file will be relative to the location of the kwcoco file itself.

The main data structure in this model is largely based on the implementation in https://github.com/cocodataset/cocoapi It uses the same efficient core indexing data structures, but in our implementation the indexing can be optionally turned off, functions are silent by default (with the exception of long running processes, which optionally show progress by default). We support helper functions that add and remove images, categories, and annotations.

The *kwcoco.CocoDataset* class is capable of dynamic addition and removal of categories, images, and annotations. Has better support for keypoints and segmentation formats than the original COCO format. Despite being written in Python, this data structure is reasonably efficient.

```
>>> import kwcoco
>>> import json
>>> # Create demo data
>>> demo = CocoDataset.demo()
>>> # could also use demo.dump / demo.dumps, but this is more explicit
>>> text = json.dumps(demo.dataset)
>>> with open('demo.json', 'w') as file:
>>>     file.write(text)

>>> # Read from disk
>>> self = CocoDataset('demo.json')

>>> # Add data
>>> cid = self.add_category('Cat')
>>> gid = self.add_image('new-img.jpg')
>>> aid = self.add_annotation(image_id=gid, category_id=cid, bbox=[0, 0, 100, 100])

>>> # Remove data
>>> self.remove_annotations([aid])
>>> self.remove_images([gid])
>>> self.remove_categories([cid])

>>> # Look at data
```

(continues on next page)

(continued from previous page)

```
>>> print(ub.repr2(self.basic_stats(), nl=1))
>>> print(ub.repr2(self.extended_stats(), nl=2))
>>> print(ub.repr2(self.boxsize_stats(), nl=3))
>>> print(ub.repr2(self.category_annotation_frequency()))


>>> # Inspect data
>>> import kwplot
>>> kwplot.autompl()
>>> self.show_image(gid=1)

>>> # Access single-item data via imgs, cats, anns
>>> cid = 1
>>> self.cats[cid]
{'id': 1, 'name': 'astronaut', 'supercategory': 'human'}

>>> gid = 1
>>> self.imgs[gid]
{'id': 1, 'file_name': 'astro.png', 'url': 'https://i.imgur.com/KXhKM72.png'}

>>> aid = 3
>>> self.anns[aid]
{'id': 3, 'image_id': 1, 'category_id': 3, 'line': [326, 369, 500, 500]}

>>> # Access multi-item data via the annots and images helper objects
>>> aids = self.index.gid_to_aids[2]
>>> annots = self.annots(aids)

>>> print('annots = {}'.format(ub.repr2(annots, nl=1, sv=1)))
annots = <Annots(num=2)>

>>> annots.lookup('category_id')
[6, 4]

>>> annots.lookup('bbox')
[[37, 6, 230, 240], [124, 96, 45, 18]]

>>> # built in conversions to efficient kwimage array DataStructures
>>> print(ub.repr2(annots.detections.data))
{
    'boxes': <Boxes(xywh,
                array([[ 37.,   6., 230., 240.],
                       [124.,  96.,  45.,  18.]], dtype=float32))>,
    'class_idxs': np.array([5, 3], dtype=np.int64),
    'keypoints': <PointsList(n=2) at 0x7f07eda33220>,
    'segmentations': <PolygonList(n=2) at 0x7f086365aa60>,
}

>>> gids = list(self.imgs.keys())
>>> images = self.images(gids)
>>> print('images = {}'.format(ub.repr2(images, nl=1, sv=1)))
images = <Images(num=3)>
```

(continues on next page)

(continued from previous page)

```
>>> images.lookup('file_name')
['astro.png', 'carl.png', 'stars.png']

>>> print('images.annots = {}'.format(images.annots))
images.annots = <AnnotGroups(n=3, m=3.7, s=3.9)>

>>> print('images.annots.cids = {!r}'.format(images.annots.cids))
images.annots.cids = [[1, 2, 3, 4, 5, 5, 5, 5, 5], [6, 4], []]
```

2.1 CocoDataset API

The following is a logical grouping of the public kwcoco.CocoDataset API attributes and methods. See the in-code documentation for further details.

2.1.1 CocoDataset classmethods (via MixinCocoExtras)

- *kwcoco.CocoDataset.coerce* - Attempt to transform the input into the intended CocoDataset.
- *kwcoco.CocoDataset.demo* - Create a toy coco dataset for testing and demo puposes
- *kwcoco.CocoDataset.random* - Creates a random CocoDataset according to distribution parameters

2.1.2 CocoDataset classmethods (via CocoDataset)

- *kwcoco.CocoDataset.from_coco_paths* - Constructor from multiple coco file paths.
- *kwcoco.CocoDataset.from_data* - Constructor from a json dictionary
- *kwcoco.CocoDataset.from_image_paths* - Constructor from a list of images paths.

2.1.3 CocoDataset slots

- kwcoco.CocoDataset.index -
- kwcoco.CocoDataset.hashid -
- kwcoco.CocoDataset.hashid_parts -
- kwcoco.CocoDataset.tag -
- kwcoco.CocoDataset.dataset -
- kwcoco.CocoDataset.bundle_dpath -
- kwcoco.CocoDataset.assets_dpath -
- kwcoco.CocoDataset.cache_dpath -

2.1.4 CocoDataset properties

- kwcoco.CocoDataset.anns -
- kwcoco.CocoDataset.cats -
- kwcoco.CocoDataset.cid_to_aids -
- kwcoco.CocoDataset.data_fpath -
- kwcoco.CocoDataset.data_root -
- *kwcoco.CocoDataset.fpath* -
- kwcoco.CocoDataset.gid_to_aids -
- kwcoco.CocoDataset.img_root -
- kwcoco.CocoDataset.imgs -
- kwcoco.CocoDataset.n_annots -
- kwcoco.CocoDataset.n_cats -
- kwcoco.CocoDataset.n_images -
- kwcoco.CocoDataset.n_videos -
- kwcoco.CocoDataset.name_to_cat -

2.1.5 CocoDataset methods (via MixinCocoAddRemove)

- *kwcoco.CocoDataset.add_annotation* - Add an annotation to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.add_annotations* - Faster less-safe multi-item alternative to add_annotation.
- *kwcoco.CocoDataset.add_category* - Adds a category
- *kwcoco.CocoDataset.add_image* - Add an image to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.add_images* - Faster less-safe multi-item alternative
- *kwcoco.CocoDataset.add_video* - Add a video to the dataset (dynamically updates the index)
- *kwcoco.CocoDataset.clear_annotations* - Removes all annotations (but not images and categories)
- *kwcoco.CocoDataset.clear_images* - Removes all images and annotations (but not categories)
- *kwcoco.CocoDataset.ensure_category* - Like `add_category()`, but returns the existing category id if it already exists instead of failing. In this case all metadata is ignored.
- *kwcoco.CocoDataset.ensure_image* - Like `add_image()`,, but returns the existing image id if it already exists instead of failing. In this case all metadata is ignored.
- *kwcoco.CocoDataset.remove_annotation* - Remove a single annotation from the dataset
- *kwcoco.CocoDataset.remove_annotation_keypoints* - Removes all keypoints with a particular category
- *kwcoco.CocoDataset.remove_annotations* - Remove multiple annotations from the dataset.
- *kwcoco.CocoDataset.remove_categories* - Remove categories and all annotations in those categories. Currently does not change any hierarchy information
- *kwcoco.CocoDataset.remove_images* - Remove images and any annotations contained by them
- *kwcoco.CocoDataset.remove_keypoint_categories* - Removes all keypoints of a particular category as well as all annotation keypoints with those ids.

- *kwcoco.CocoDataset.remove_videos* - Remove videos and any images / annotations contained by them
- *kwcoco.CocoDataset.set_annotation_category* - Sets the category of a single annotation

2.1.6 CocoDataset methods (via MixinCocoObjects)

- *kwcoco.CocoDataset.annots* - Return vectorized annotation objects
- *kwcoco.CocoDataset.categories* - Return vectorized category objects
- *kwcoco.CocoDataset.images* - Return vectorized image objects
- *kwcoco.CocoDataset.videos* - Return vectorized video objects

2.1.7 CocoDataset methods (via MixinCocoStats)

- *kwcoco.CocoDataset.basic_stats* - Reports number of images, annotations, and categories.
- *kwcoco.CocoDataset.boxsize_stats* - Compute statistics about bounding box sizes.
- *kwcoco.CocoDataset.category_annotation_frequency* - Reports the number of annotations of each category
- *kwcoco.CocoDataset.category_annotation_type_frequency* - Reports the number of annotations of each type for each category
- *kwcoco.CocoDataset.conform* - Make the COCO file conform a stricter spec, infers attibutes where possible.
- *kwcoco.CocoDataset.extended_stats* - Reports number of images, annotations, and categories.
- *kwcoco.CocoDataset.find_representative_images* - Find images that have a wide array of categories. Attempt to find the fewest images that cover all categories using images that contain both a large and small number of annotations.
- *kwcoco.CocoDataset.keypoint_annotation_frequency* -
- *kwcoco.CocoDataset.stats* - This function corresponds to **:module:`kwcoco.cli.coco_stats`**.
- *kwcoco.CocoDataset.validate* - Performs checks on this coco dataset.

2.1.8 CocoDataset methods (via MixinCocoAccessors)

- *kwcoco.CocoDataset.category_graph* - Construct a networkx category hierarchy
- *kwcoco.CocoDataset.delayed_load* - Experimental method
- *kwcoco.CocoDataset.get_auxiliary_fpath* - Returns the full path to auxiliary data for an image
- *kwcoco.CocoDataset.get_image_fpath* - Returns the full path to the image
- *kwcoco.CocoDataset.keypoint_categories* - Construct a consistent CategoryTree representation of keypoint classes
- *kwcoco.CocoDataset.load_annot_sample* - Reads the chip of an annotation. Note this is much less efficient than using a sampler, but it doesn't require disk cache.
- *kwcoco.CocoDataset.load_image* - Reads an image from disk and
- *kwcoco.CocoDataset.object_categories* - Construct a consistent CategoryTree representation of object classes

2.1.9 CocoDataset methods (via CocoDataset)

- *kwcoco.CocoDataset.copy* - Deep copies this object
- *kwcoco.CocoDataset.dump* - Writes the dataset out to the json format
- *kwcoco.CocoDataset.dumps* - Writes the dataset out to the json format
- *kwcoco.CocoDataset.subset* - Return a subset of the larger coco dataset by specifying which images to port. All annotations in those images will be taken.
- *kwcoco.CocoDataset.union* - Merges multiple *CocoDataset* items into one. Names and associations are retained, but ids may be different.
- *kwcoco.CocoDataset.view_sql* - Create a cached SQL interface to this dataset suitable for large scale multiprocessing use cases.

2.1.10 CocoDataset methods (via MixinCocoExtras)

- *kwcoco.CocoDataset.corrupted_images* - Check for images that don't exist or can't be opened
- *kwcoco.CocoDataset.missing_images* - Check for images that don't exist
- *kwcoco.CocoDataset.rename_categories* - Rename categories with a potentially coarser categorization.
- *kwcoco.CocoDataset.reroot* - Rebase image/data paths onto a new image/data root.

2.1.11 CocoDataset methods (via MixinCocoDraw)

- *kwcoco.CocoDataset.draw_image* - Use kwimage to draw all annotations on an image and return the pixels as a numpy array.
- *kwcoco.CocoDataset.imread* - Loads a particular image
- *kwcoco.CocoDataset.show_image* - Use matplotlib to show an image with annotations overlaid

2.2 Subpackages

2.2.1 kwcoco.cli

Submodules

kwcoco.cli.__main__

Module Contents

Functions

main(cmdline=True, **kw)	kw = dict(command='stats')

kwcoco.cli.__main__.**main**(*cmdline=True*, ***kw*)
kw = dict(command='stats') cmdline = False

kwcoco.cli.coco_conform

Module Contents

Classes

CocoConformCLI

Attributes

_CLI

class kwcoco.cli.coco_conform.**CocoConformCLI**

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Make the COCO file conform to the spec.

Populates inferable information such as image size, annotation area, etc.

epilog = Multiline-String

```
        Example Usage:
            kwcoco conform --help
            kwcoco conform --src=special:shapes8 --dst conformed.json
```

default

name = conform

classmethod main(*cls*, *cmdline=True*, ***kw*)

Example

```
>>> # xdoctest: +SKIP
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoConformCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_conform.**_CLI**

kwcoco.cli.coco_eval

Module Contents

Classes

CocoEvalCLI

Attributes

_CLI

class kwcoco.cli.coco_eval.**CocoEvalCLI**

name = eval

CLIConfig

classmethod main(*cls*, *cmdline=True*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import ubelt as ub
>>> from kwcoco.cli.coco_eval import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> from os.path import join
>>> import kwcoco
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/eval')
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> true_dset.fpath = join(dpath, 'true.mscoco.json')
>>> pred_dset.fpath = join(dpath, 'pred.mscoco.json')
>>> true_dset.dump(true_dset.fpath)
>>> pred_dset.dump(pred_dset.fpath)
>>> draw = False  # set to false for faster tests
>>> CocoEvalCLI.main(
>>>     true_dataset=true_dset.fpath,
>>>     pred_dataset=pred_dset.fpath,
>>>     draw=draw)
```

kwcoco.cli.coco_eval._CLI

kwcoco.cli.coco_grab

Module Contents

Classes

CocoGrabCLI	

Attributes

_CLI

class kwcoco.cli.coco_grab.**CocoGrabCLI**

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Grab standard datasets.

Example

kwcoco grab cifar10 camvid

default

name = grab

classmethod main(*cls*, *cmdline=True*, ***kw*)

kwcoco.cli.coco_grab._CLI

kwcoco.cli.coco_modify_categories

Module Contents

Classes

CocoModifyCatsCLI	Remove, rename, or coarsen categories.

Attributes

_CLI

class kwcoco.cli.coco_modify_categories.**CocoModifyCatsCLI**

Remove, rename, or coarsen categories.

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Rename or remove categories

epilog = Multiline-String

```
        Example Usage:
            kwcoco modify_categories --help
            kwcoco modify_categories --src=special:shapes8 --dst␣
↪modcats.json
            kwcoco modify_categories --src=special:shapes8 --dst␣
↪modcats.json --rename eff:F,star:sun
            kwcoco modify_categories --src=special:shapes8 --dst␣
↪modcats.json --remove eff,star
            kwcoco modify_categories --src=special:shapes8 --dst␣
↪modcats.json --keep eff,

            kwcoco modify_categories --src=special:shapes8 --dst␣
↪modcats.json --keep=[] --keep_annots=True
```

default

name = modify_categories

classmethod main(*cls*, *cmdline=True*, ***kw*)

Example

```
>>> # xdoctest: +SKIP
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoModifyCatsCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_modify_categories.**_CLI**

kwcoco.cli.coco_reroot

Module Contents

Classes

CocoRerootCLI

Attributes

_CLI

class kwcoco.cli.coco_reroot.**CocoRerootCLI**

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Reroot image paths onto a new image root.

Modify the root of a coco dataset such to either make paths relative to a new root or make paths absolute.

Todo:

- [ ] Evaluate that all tests cases work

epilog = Multiline-String

```
        Example Usage:
            kwcoco reroot --help
            kwcoco reroot --src=special:shapes8 --dst rerooted.json
            kwcoco reroot --src=special:shapes8 --new_prefix=foo --
↪check=True --dst rerooted.json
```

default

name = reroot

classmethod main(*cls*, *cmdline=True*, ***kw*)

Example

```
>>> # xdoctest: +SKIP
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoRerootCLI
>>> cls.main(cmdline, **kw)
```

Ignore: python ~/code/kwcoco/kwcoco/cli/coco_reroot.py –src=$HOME/code/bioharn/fast_test/deep_training/training_truth
–dst=$HOME/code/bioharn/fast_test/deep_training/training_truth2.json
–new_prefix=/home/joncrall/code/bioharn/fast_test/training_data –old_prefix=/run/media/matt/Storage/TEST/training_

python ~/code/kwcoco/kwcoco/cli/coco_reroot.py –src=$HOME/code/bioharn/fast_test/deep_training/validation_truth.j
–dst=$HOME/code/bioharn/fast_test/deep_training/validation_truth2.json
–new_prefix=/home/joncrall/code/bioharn/fast_test/training_data –old_prefix=/run/media/matt/Storage/TEST/training_

cmdline = ‘’’ –src=$HOME/code/bioharn/fast_test/deep_training/training_truth.json
–dst=$HOME/code/bioharn/fast_test/deep_training/training_truth2.json
–new_prefix=/home/joncrall/code/bioharn/fast_test/training_data
–old_prefix=/run/media/matt/Storage/TEST/training_data

‘’’ /run/media/matt/Storage/TEST/training_data –check=True –dst rerooted.json

kwcoco.cli.coco_reroot.**_CLI**

kwcoco.cli.coco_show

Module Contents

Classes

CocoShowCLI

Attributes

_CLI

class kwcoco.cli.coco_show.**CocoShowCLI**

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Visualize a COCO image using matplotlib or opencv, optionally writing it to disk

epilog = Multiline-String

```
1       Example Usage:
```

(continues on next page)

(continued from previous page)

```
            kwcoco show --help
            kwcoco show --src=special:shapes8 --gid=1
            kwcoco show --src=special:shapes8 --gid=1 --dst out.png
```

default

name = show

classmethod main(*cls*, *cmdline=True*, ***kw*)

Todo:

- [ ] Visualize auxiliary data

Example

```
>>> # xdoctest: +SKIP
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoShowCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_show.**_CLI**

kwcoco.cli.coco_split

Module Contents

Classes

CocoSplitCLI

Attributes

_CLI

class kwcoco.cli.coco_split.**CocoSplitCLI**

Bases: object

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Split a single COCO dataset into two sub-datasets.

default

epilog = Multiline-String

```
        Example Usage:
            kwcoco split --src special:shapes8 --dst1=learn.mscoco.
↪json --dst2=test.mscoco.json --factor=3 --rng=42
```

name = split

classmethod main(*cls*, *cmdline=True*, ***kw*)

Example

```
>>> kw = {'src': 'special:shapes8',
>>>       'dst1': 'train.json', 'dst2': 'test.json'}
>>> cmdline = False
>>> cls = CocoSplitCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_split.**_CLI**

kwcoco.cli.coco_stats

Module Contents

Classes

CocoStatsCLI

Attributes

_CLI

class kwcoco.cli.coco_stats.**CocoStatsCLI**

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Compute summary statistics about a COCO dataset

default

epilog = Multiline-String

```
        Example Usage:
            kwcoco stats --src=special:shapes8
            kwcoco stats --src=special:shapes8 --boxes=True
```

name = stats

classmethod main(*cls*, *cmdline=True*, ***kw*)

Example

```
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoStatsCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_stats.**_CLI**

kwcoco.cli.coco_subset

Module Contents

Classes

CocoSubsetCLI

Functions

query_subset(dset, config)

Example

Attributes

_CLI

class kwcoco.cli.coco_subset.**CocoSubsetCLI**

Bases: object

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Take a subset of this dataset and write it to a new file

default

epilog = Multiline-String

```
        Example Usage:
            kwcoco subset --src special:shapes8 --dst=foo.kwcoco.json

            # Take only the even image-ids
            kwcoco subset --src special:shapes8 --dst=foo-even.kwcoco.
↪json --select_images '.id % 2 == 0'

            # Take only the videos where the name ends with 2
            kwcoco subset --src special:vidshapes8 --dst=vidsub.
↪kwcoco.json --select_videos '.name | endswith("2")'
```

name = subset

classmethod main(*cls*, *cmdline=True*, ***kw*)

Example

```
>>> kw = {'src': 'special:shapes8',
>>>       'dst': 'subset.json', 'include_categories': 'superstar'}
>>> cmdline = False
>>> cls = CocoSubsetCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_subset.**query_subset**(*dset*, *config*)

Example

```
>>> # xdoctest: +REQUIRES(module:jq)
>>> from kwcoco.cli.coco_subset import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> assert dset.n_images == 3
>>> #
>>> config = CocoSubsetCLI.CLIConfig({'select_images': '.id < 3'})
>>> new_dset = query_subset(dset, config)
>>> assert new_dset.n_images == 2
>>> #
>>> config = CocoSubsetCLI.CLIConfig({'select_images': '.file_name | test(".*.png")
↪'})
>>> new_dset = query_subset(dset, config)
>>> assert all(n.endswith('.png') for n in new_dset.images().lookup('file_name'))
>>> assert new_dset.n_images == 2
>>> #
>>> config = CocoSubsetCLI.CLIConfig({'select_images': '.file_name | test(".*.png")␣
↪| not'})
>>> new_dset = query_subset(dset, config)
>>> assert not any(n.endswith('.png') for n in new_dset.images().lookup('file_name
↪'))
>>> assert new_dset.n_images == 1
```

(continues on next page)

(continued from previous page)

```
>>> #
>>> config = CocoSubsetCLI.CLIConfig({'select_images': '.id < 3 and (.file_name |␣
↪test(".*.png"))'})
>>> new_dset = query_subset(dset, config)
>>> assert new_dset.n_images == 1
>>> #
>>> config = CocoSubsetCLI.CLIConfig({'select_images': '.id < 3 or (.file_name |␣
↪test(".*.png"))'})
>>> new_dset = query_subset(dset, config)
>>> assert new_dset.n_images == 3
```

Example

```
>>> # xdoctest: +REQUIRES(module:jq)
>>> from kwcoco.cli.coco_subset import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8')
>>> assert dset.n_videos == 8
>>> assert dset.n_images == 16
>>> config = CocoSubsetCLI.CLIConfig({'select_videos': '.name == "toy_video_3"'})
>>> new_dset = query_subset(dset, config)
>>> assert new_dset.n_images == 2
>>> assert new_dset.n_videos == 1
```

kwcoco.cli.coco_subset._CLI

kwcoco.cli.coco_toydata

Module Contents

Classes

CocoToyDataCLI

Attributes

_CLI

class kwcoco.cli.coco_toydata.**CocoToyDataCLI**

Bases: object

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Create COCO toydata

default

epilog = Multiline-String

```
        Example Usage:
            kwcoco toydata --key=shapes8 --dst=toydata.kwcoco.json

            kwcoco toydata --key=shapes8 --bundle_dpath=my_test_
↪bundle_v1
            kwcoco toydata --key=shapes8 --bundle_dpath=my_test_
↪bundle_v1

            kwcoco toydata                 --key=shapes8
↪   --dst=./shapes8.kwcoco/dataset.kwcoco.json

        TODO:
            - [ ] allow specification of images directory
```

name = toydata

classmethod main(*cls*, *cmdline=True*, ***kw*)

Example

```
>>> kw = {'key': 'shapes8', 'dst': 'test.json'}
>>> cmdline = False
>>> cls = CocoToyDataCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_toydata.**_CLI**

kwcoco.cli.coco_union

Module Contents

Classes

CocoUnionCLI

Attributes

_CLI

class kwcoco.cli.coco_union.**CocoUnionCLI**
Bases: object

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)
Bases: scriptconfig.Config

Combine multiple COCO datasets into a single merged dataset.

default

epilog = Multiline-String

```
        Example Usage:
            kwcoco union --src special:shapes8 special:shapes1 --
↪dst=combo.mscoco.json
```

name = union

classmethod main(*cls*, *cmdline=True*, ***kw*)

Example

```
>>> kw = {'src': ['special:shapes8', 'special:shapes1']}
>>> cmdline = False
>>> cls = CocoUnionCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_union.**_CLI**

kwcoco.cli.coco_validate

Module Contents

Classes

CocoValidateCLI

Attributes

_CLI

class kwcoco.cli.coco_validate.**CocoValidateCLI**

class CLIConfig(*data=None*, *default=None*, *cmdline=False*)
Bases: scriptconfig.Config

Validate that a coco file conforms to the json schema, that assets exist, and potentially fix corrupted assets by removing them.

default

epilog = Multiline-String

```
        Example Usage:
            kwcoco toydata --dst foo.json --key=special:shapes8
            kwcoco validate --src=foo.json --corrupted=True
```

name = validate

classmethod main(*cls*, *cmdline=True*, ***kw*)

Example

```
>>> from kwcoco.cli.coco_validate import *  # NOQA
>>> kw = {'src': 'special:shapes8'}
>>> cmdline = False
>>> cls = CocoValidateCLI
>>> cls.main(cmdline, **kw)
```

kwcoco.cli.coco_validate.**_CLI**

2.2.2 kwcoco.data

Submodules

kwcoco.data.grab_camvid

Downloads the CamVid data if necessary, and converts it to COCO.

Module Contents

Functions

_devcheck_sample_full_image()	
_devcheck_load_sub_image()	
grab_camvid_train_test_val_splits(coco_dset, mode='segnet')	
grab_camvid_sampler()	Grab a kwcoco.CocoSampler object for the CamVid dataset.
grab_coco_camvid()	
	Example
grab_raw_camvid()	Grab the raw camvid data.
rgb_to_cid(r, g, b)	
cid_to_rgb(cid)	
convert_camvid_raw_to_coco(camvid_raw_info)	Converts the raw camvid format to an MSCOCO based format, (which lets use
_define_camvid_class_hierarcy(dset)	
main()	Dump the paths to the coco file to stdout

kwcoco.data.grab_camvid.**_devcheck_sample_full_image**()

kwcoco.data.grab_camvid.**_devcheck_load_sub_image**()

kwcoco.data.grab_camvid.**grab_camvid_train_test_val_splits**(*coco_dset*, *mode='segnet'*)

kwcoco.data.grab_camvid.**grab_camvid_sampler**()

Grab a kwcoco.CocoSampler object for the CamVid dataset.

Returns sampler

Return type kwcoco.CocoSampler

Example

```
>>> # xdoctest: +REQUIRES(--download)
>>> sampler = grab_camvid_sampler()
>>> print('sampler = {!r}'.format(sampler))
>>> # sampler.load_sample()
>>> for gid in ub.ProgIter(sampler.image_ids, desc='load image'):
>>>     img = sampler.load_image(gid)
```

kwcoco.data.grab_camvid.**grab_coco_camvid**()

Example

```
>>> # xdoctest: +REQUIRES(--download)
>>> dset = grab_coco_camvid()
>>> print('dset = {!r}'.format(dset))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> plt.clf()
>>> dset.show_image(gid=1)
```

Ignore: import xdev gid_list = list(dset.imgs) for gid in xdev.InteractiveIter(gid_list):

dset.show_image(gid) xdev.InteractiveIter.draw()

kwcoco.data.grab_camvid.**grab_raw_camvid**()
Grab the raw camvid data.

kwcoco.data.grab_camvid.**rgb_to_cid**(*r*, *g*, *b*)

kwcoco.data.grab_camvid.**cid_to_rgb**(*cid*)

kwcoco.data.grab_camvid.**convert_camvid_raw_to_coco**(*camvid_raw_info*)
Converts the raw camvid format to an MSCOCO based format, (which lets use use kwcoco's COCO backend).

Example

```
>>> # xdoctest: +REQUIRES(--download)
>>> camvid_raw_info = grab_raw_camvid()
>>> # test with a reduced set of data
>>> del camvid_raw_info['img_paths'][2:]
>>> del camvid_raw_info['mask_paths'][2:]
>>> dset = convert_camvid_raw_to_coco(camvid_raw_info)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> kwplot.figure(fnum=1, pnum=(1, 2, 1))
>>> dset.show_image(gid=1)
>>> kwplot.figure(fnum=1, pnum=(1, 2, 2))
>>> dset.show_image(gid=2)
```

kwcoco.data.grab_camvid.**_define_camvid_class_hierarcy**(*dset*)

kwcoco.data.grab_camvid.**main**()
Dump the paths to the coco file to stdout

By default these will go to in the path: ~/.cache/kwcoco/camvid/camvid-master

The four files will be: ~/.cache/kwcoco/camvid/camvid-master/camvid-full.mscoco.json ~/.cache/kwcoco/camvid/camvid-master/camvid-train.mscoco.json ~/.cache/kwcoco/camvid/camvid-master/camvid-vali.mscoco.json ~/.cache/kwcoco/camvid/camvid-master/camvid-test.mscoco.json

kwcoco.data.grab_cifar

Downloads and converts CIFAR 10 and CIFAR 100 to kwcoco format

Module Contents

Functions

_convert_cifar_x(dpath, cifar_dset, cifar_name, classes)	
convert_cifar10(dpath=None)	
convert_cifar100(dpath=None)	
main()	

kwcoco.data.grab_cifar.**_convert_cifar_x**(*dpath*, *cifar_dset*, *cifar_name*, *classes*)

kwcoco.data.grab_cifar.**convert_cifar10**(*dpath=None*)

kwcoco.data.grab_cifar.**convert_cifar100**(*dpath=None*)

kwcoco.data.grab_cifar.**main**()

kwcoco.data.grab_datasets

[ ] HMDB: a large human motion database - https://serre-lab.clps.brown.edu/resource/hmdb-a-large-human-motion-database/

[ ] https://paperswithcode.com/dataset/imagenet

[ ] https://paperswithcode.com/dataset/coco

[ ] https://paperswithcode.com/dataset/fashion-mnist

[ ] https://paperswithcode.com/dataset/visual-question-answering

[ ] https://paperswithcode.com/dataset/lfw

[ ] https://paperswithcode.com/dataset/lsun

[ ] https://paperswithcode.com/dataset/ava

[ ] https://paperswithcode.com/dataset/activitynet

[ ] https://paperswithcode.com/dataset/clevr:

kwcoco.data.grab_domainnet

References

http://ai.bu.edu/M3SDA/#dataset

Module Contents

Functions

grab_domain_net()	

kwcoco.data.grab_domainnet.**grab_domain_net**()

Todo:

- [ ] Allow the user to specify the download directory, generalize this

pattern across the data grab scripts.

kwcoco.data.grab_spacenet

References

https://medium.com/the-downlinq/the-spacenet-7-multi-temporal-urban-development-challenge-algorithmic-baseline-4515ec9bd9fe
https://arxiv.org/pdf/2102.11958.pdf https://spacenet.ai/sn7-challenge/

Module Contents

Classes

Archive	Abstraction over zipfile and tarfile

Functions

unarchive_file(archive_fpath, output_dpath='.', verbose=1, overwrite=True)	
grab_spacenet7(data_dpath)	

References

continues on next page

Table 31 – continued from previous page

convert_spacenet_to_kwcoco(extract_dpath, coco_fpath)	Converts the raw SpaceNet7 dataset to kwcoco
main()	

class kwcoco.data.grab_spacenet.**Archive**(*fpath*, *mode='r'*)

Bases: object

Abstraction over zipfile and tarfile

Example

```
>>> from kwcoco.data.grab_spacenet import *  # NOQA
>>> from os.path import join
>>> dpath = ub.ensure_app_cache_dir('ubelt', 'tests', 'archive')
>>> ub.delete(dpath)
>>> dpath = ub.ensure_app_cache_dir(dpath)
>>> import pathlib
>>> dpath = pathlib.Path(dpath)
>>> #
>>> #
>>> mode = 'w'
>>> self1 = Archive(str(dpath / 'demo.zip'), mode=mode)
>>> self2 = Archive(str(dpath / 'demo.tar.gz'), mode=mode)
>>> #
>>> open(dpath / 'data_1only.txt', 'w').write('bazbzzz')
>>> open(dpath / 'data_2only.txt', 'w').write('buzzz')
>>> open(dpath / 'data_both.txt', 'w').write('foobar')
>>> #
>>> self1.add(dpath / 'data_both.txt')
>>> self1.add(dpath / 'data_1only.txt')
>>> #
>>> self2.add(dpath / 'data_both.txt')
>>> self2.add(dpath / 'data_2only.txt')
>>> #
>>> self1.close()
>>> self2.close()
>>> #
>>> self1 = Archive(str(dpath / 'demo.zip'), mode='r')
>>> self2 = Archive(str(dpath / 'demo.tar.gz'), mode='r')
>>> #
>>> extract_dpath = ub.ensuredir(str(dpath / 'extracted'))
>>> extracted1 = self1.extractall(extract_dpath)
>>> extracted2 = self2.extractall(extract_dpath)
>>> for fpath in extracted2:
>>>     print(open(fpath, 'r').read())
>>> for fpath in extracted1:
>>>     print(open(fpath, 'r').read())
```

__iter__(*self*)

add(*self*, *fpath*, *arcname=None*)

close(*self*)

__enter__(*self*)

__exit__(*self*, **args*)

extractall(*self*, *output_dpath='.'*, *verbose=1*, *overwrite=True*)

kwcoco.data.grab_spacenet.**unarchive_file**(*archive_fpath*, *output_dpath='.'*, *verbose=1*, *overwrite=True*)

kwcoco.data.grab_spacenet.**grab_spacenet7**(*data_dpath*)

References

https://spacenet.ai/sn7-challenge/

Requires: awscli

Ignore: mkdir -p $HOME/.cache/kwcoco/data/spacenet/archives cd $HOME/.cache/kwcoco/data/spacenet/archives

Requires an AWS account export AWS_PROFILE=joncrall_at_kitware

aws s3 cp s3://spacenet-dataset/spacenet/SN7_buildings/tarballs/SN7_buildings_train.tar.gz . aws s3 cp s3://spacenet-dataset/spacenet/SN7_buildings/tarballs/SN7_buildings_train_csvs.tar.gz . aws s3 cp s3://spacenet-dataset/spacenet/SN7_buildings/tarballs/SN7_buildings_test_public.tar.gz .

kwcoco.data.grab_spacenet.**convert_spacenet_to_kwcoco**(*extract_dpath*, *coco_fpath*)
Converts the raw SpaceNet7 dataset to kwcoco

Notes

- The “train” directory contains 60 “videos” representing a region over

time.

- **Each “video” directory contains :**
  - images - unmasked images
  - images_masked - images with masks applied
  - labels - geojson polys in wgs84?
  - labels_match - geojson polys in wgs84 with track ids?
  - labels_match_pix - geojson polys in pixels with track ids?
  - UDM_masks - unusable data masks (binary data corresponding with an image, may not exist)

File names appear like: “global_monthly_2018_01_mosaic_L15-1538E-1163N_6154_3539_13”

Ignore: dpath = pathlib.Path(“/home/joncrall/data/dvc-repos/smart_watch_dvc/extern/spacenet/”) extract_dpath = dpath / ‘extracted’ coco_fpath = dpath / ‘spacenet7.kwcoco.json’

kwcoco.data.grab_spacenet.**main**()

kwcoco.data.grab_voc

Module Contents

Functions

__torrent_voc()	Requires:
convert_voc_to_coco(dpath=None)	
_convert_voc_split(devkit_dpath, classes, split, year, root)	split, year = 'train', 2012
_read_split_paths(devkit_dpath, split, year)	split = 'train'
ensure_voc_data(dpath=None, force=False, years=[2007, 2012])	Download the Pascal VOC data if it does not already exist.
ensure_voc_coco(dpath=None)	Download the Pascal VOC data and convert it to coco, if it does exit.
main()	

kwcoco.data.grab_voc.**__torrent_voc**()

Requires: pip install deluge pip install python-libtorrent-bin

References

https://academictorrents.com/details/f6ddac36ac7ae2ef79dc72a26a065b803c9c7230

Todo:

- [ ] Is there a pythonic way to download a torrent programatically?

kwcoco.data.grab_voc.**convert_voc_to_coco**(*dpath=None*)

kwcoco.data.grab_voc.**_convert_voc_split**(*devkit_dpath*, *classes*, *split*, *year*, *root*)
split, year = 'train', 2012 split, year = 'train', 2007

kwcoco.data.grab_voc.**_read_split_paths**(*devkit_dpath*, *split*, *year*)
split = 'train' self = VOCDataset('test') year = 2007 year = 2012

kwcoco.data.grab_voc.**ensure_voc_data**(*dpath=None*, *force=False*, *years=[2007, 2012]*)
Download the Pascal VOC data if it does not already exist.

Note:

- [ ] These URLS seem to be dead

Example

```
>>> # xdoctest: +REQUIRES(--download)
>>> devkit_dpath = ensure_voc_data()
```

kwcoco.data.grab_voc.**ensure_voc_coco**(*dpath=None*)

Download the Pascal VOC data and convert it to coco, if it does exit.

Parameters **dpath** (*str*) – download directory. Defaults to "~/data/VOC".

Returns

mapping from dataset tags to coco file paths. The original datasets have keys prefixed with underscores. The standard splits keys are train, vali, and test.

Return type Dict[str, str]

kwcoco.data.grab_voc.**main**()

2.2.3 kwcoco.demo

Submodules

kwcoco.demo.boids

Module Contents

Classes

Boids	Efficient numpy based backend for generating boid positions.

Functions

clamp_mag(vec, mag, axis=None)	vec = np.random.rand(10, 2)
triu_condense_multi_index(multi_index, dims, symetric=False)	Like np.ravel_multi_index but returns positions in an upper triangular
_spatial_index_scratch()	Ignore:
closest_point_on_line_segment(pts, e1, e2)	Finds the closet point from p on line segment (e1, e2)
_pygame_render_boids()	Fast and responsive BOID rendering. This is an easter egg.
_yeah_boid()	

class kwcoco.demo.boids.**Boids**(*num*, *dims=2*, *rng=None*, ***kwargs*)

Bases: ubelt.NiceRepr

Efficient numpy based backend for generating boid positions.

BOID = bird-oid object

References

https://www.youtube.com/watch?v=mhjuuHl6qHM https://medium.com/better-programming/boids-simulating-birds-flock-behavior-in-python-9fff99375118 https://en.wikipedia.org/wiki/Boids

Example

```
>>> from kwcoco.demo.boids import *  # NOQA
>>> num_frames = 10
>>> num_objects = 3
>>> rng = None
>>> self = Boids(num=num_objects, rng=rng).initialize()
>>> paths = self.paths(num_frames)
>>> #
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> from mpl_toolkits.mplot3d import Axes3D  # NOQA
>>> ax = plt.gca(projection='3d')
>>> ax.cla()
>>> #
>>> for path in paths:
>>>     time = np.arange(len(path))
>>>     ax.plot(time, path.T[0] * 1, path.T[1] * 1, ',-')
>>> ax.set_xlim(0, num_frames)
>>> ax.set_ylim(-.01, 1.01)
>>> ax.set_zlim(-.01, 1.01)
>>> ax.set_xlabel('time')
>>> ax.set_ylabel('u-pos')
>>> ax.set_zlabel('v-pos')
>>> kwplot.show_if_requested()
```

import xdev _ = xdev.profile_now(self.compute_forces)() _ = xdev.profile_now(self.update_neighbors)()

Ignore: self = Boids(num=5, rng=0).initialize() self.pos

fig = kwplot.figure(fnum=10, do_clf=True) ax = fig.gca()

verts = np.array([[0, 0], [1, 0], [0.5, 2]]) com = verts.mean(axis=0) verts = (verts - com) * 0.02

import kwimage poly = kwimage.Polygon(exterior=verts)

def rotate(poly, theta): **sin_** = np.sin(theta) **cos_** = np.cos(theta) **rot_** = np.array(((**cos_**, -**sin_**), (**sin_**, **cos_**),))

return poly.warp(**rot_**)

for _ in xdev.InteractiveIter(list(range(10000))): self.step() ax.cla() import math for rx in range(len(self.pos)):

x, y = self.pos[rx] dx, dy = (self.vel[rx] / np.linalg.norm(self.vel[rx], axis=0))

theta = (np.arctan2(dy, dx) - math.tau / 4) boid_poly = rotate(poly, theta).translate(self.pos[rx]) color = 'red' if rx == 0 else 'blue' boid_poly.draw(ax=ax, color=color)

tip = boid_poly.data['exterior'].data[2] tx, ty = tip

s = 100.0 vel = self.vel[rx] acc = self.acc[rx] com = self.acc[rx]

spsteer = self.sep_steering[rx] cmsteer = self.com_steering[rx] alsteer = self.align_steering[rx] avsteer = self.avoid_steering[rx]

plt.arrow(tip[0], tip[1], s * vel[0], s * vel[1], color='green') plt.arrow(tip[0], tip[1], s * acc[1], s * acc[1], color='purple')

plt.arrow(tip[0], tip[1], s * spsteer[0], s * spsteer[1], color='dodgerblue') plt.arrow(tip[0], tip[1], s * cmsteer[0], s * cmsteer[1], color='orange') plt.arrow(tip[0], tip[1], s * alsteer[0], s * alsteer[1], color='pink') plt.arrow(tip[0], tip[1], s * avsteer[0], s * avsteer[1], color='black')

ax.set_xlim(0, 1) ax.set_ylim(0, 1) xdev.InteractiveIter.draw()

rx = 0

__nice__(*self*)

initialize(*self*)

update_neighbors(*self*)

compute_forces(*self*)

boundary_conditions(*self*)

step(*self*)
Update positions, velocities, and accelerations

paths(*self*, *num_steps*)

kwcoco.demo.boids.**clamp_mag**(*vec*, *mag*, *axis=None*)
vec = np.random.rand(10, 2) mag = 1.0 axis = 1 new_vec = clamp_mag(vec, mag, axis) np.linalg.norm(new_vec, axis=axis)

kwcoco.demo.boids.**triu_condense_multi_index**(*multi_index*, *dims*, *symetric=False*)
Like np.ravel_multi_index but returns positions in an upper triangular condensed square matrix

Examples

multi_index (Tuple[ArrayLike]): indexes for each dimension into the square matrix

dims (Tuple[int]): shape of each dimension in the square matrix (should all be the same)

symetric (bool): if True, converts lower triangular indices to their upper triangular location. This may cause a copy to occur.

References

https://stackoverflow.com/a/36867493/887074 https://numpy.org/doc/stable/reference/generated/numpy.ravel_multi_index.html#numpy.ravel_multi_index

Examples

```
>>> dims = (3, 3)
>>> symetric = True
>>> multi_index = (np.array([0, 0, 1]), np.array([1, 2, 2]))
>>> condensed_idxs = triu_condense_multi_index(multi_index, dims, symetric=symetric)
>>> assert condensed_idxs.tolist() == [0, 1, 2]
```

```
>>> n = 7
>>> symetric = True
>>> multi_index = np.triu_indices(n=n, k=1)
>>> condensed_idxs = triu_condense_multi_index(multi_index, [n] * 2,␣
↪symetric=symetric)
>>> assert condensed_idxs.tolist() == list(range(n * (n - 1) // 2))
>>> from scipy.spatial.distance import pdist, squareform
>>> square_mat = np.zeros((n, n))
>>> conden_mat = squareform(square_mat)
>>> conden_mat[condensed_idxs] = np.arange(len(condensed_idxs)) + 1
>>> square_mat = squareform(conden_mat)
>>> print('square_mat =\n{}'.format(ub.repr2(square_mat, nl=1)))
```

```
>>> n = 7
>>> symetric = True
>>> multi_index = np.tril_indices(n=n, k=-1)
>>> condensed_idxs = triu_condense_multi_index(multi_index, [n] * 2,␣
↪symetric=symetric)
>>> assert sorted(condensed_idxs.tolist()) == list(range(n * (n - 1) // 2))
>>> from scipy.spatial.distance import pdist, squareform
>>> square_mat = np.zeros((n, n))
>>> conden_mat = squareform(square_mat, checks=False)
>>> conden_mat[condensed_idxs] = np.arange(len(condensed_idxs)) + 1
>>> square_mat = squareform(conden_mat)
>>> print('square_mat =\n{}'.format(ub.repr2(square_mat, nl=1)))
```

Ignore:

```
>>> import xdev
>>> n = 30
>>> symetric = True
>>> multi_index = np.triu_indices(n=n, k=1)
>>> condensed_idxs = xdev.profile_now(triu_condense_multi_index)(multi_index,␣
↪[n] * 2)
```

Ignore: # Numba helps here when ub.allsame is gone from numba import jit triu_condense_multi_index2 = jit(nopython=True)(triu_condense_multi_index) triu_condense_multi_index2 = jit()(triu_condense_multi_index) triu_condense_multi_index2(multi_index, [n] * 2) %timeit triu_condense_multi_index(multi_index, [n] * 2) %timeit triu_condense_multi_index2(multi_index, [n] * 2)

kwcoco.demo.boids.**_spatial_index_scratch**()

Ignore: !pip install git+git://github.com/carsonfarmer/fastpair.git

from fastpair import FastPair fp = FastPair() fp.build(list(map(tuple, self.pos.tolist()))) #

!pip install pyqtree from pyqtree import Index spindex = Index(bbox=(0, 0, 1., 1.))

this example assumes you have a list of items with bbox attribute for item in items:

spindex.insert(item, item.bbox)

https://stackoverflow.com/questions/41946007/efficient-and-well-explained-implementation-of-a-quadtree-for-2d-collision-

!pip install grispy import grispy

index = grispy.GriSPy(data=self.pos) index.bubble_neighbors(self.pos, distance_upper_bound=0.1)

kwcoco.demo.boids.**closest_point_on_line_segment**(*pts*, *e1*, *e2*)

Finds the closet point from p on line segment (e1, e2)

Parameters

- **pts** (*ndarray*) – xy points [Nx2]
- **e1** (*ndarray*) – the first xy endpoint of the segment
- **e2** (*ndarray*) – the second xy endpoint of the segment

Returns pt_on_seg - the closest xy point on (e1, e2) from ptp

Return type ndarray

References

http://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line http://stackoverflow.com/questions/849211/shortest-distance-between-a-point-and-a-line-segment

Example

```
>>> # ENABLE_DOCTEST
>>> from kwcoco.demo.boids import *  # NOQA
>>> verts = np.array([[ 21.83012702,  13.16987298],
>>>                   [ 16.83012702,  21.83012702],
>>>                   [  8.16987298,  16.83012702],
>>>                   [ 13.16987298,   8.16987298],
>>>                   [ 21.83012702,  13.16987298]])
>>> rng = np.random.RandomState(0)
>>> pts = rng.rand(64, 2) * 20 + 5
>>> e1, e2 = verts[0:2]
>>> closest_point_on_line_segment(pts, e1, e2)
```

Ignore: from numba import jit closest_point_on_line_segment2 = jit(closest_point_on_line_segment) closest_point_on_line_segment2(pts, e1, e2) %timeit closest_point_on_line_segment(pts, e1, e2) %timeit closest_point_on_line_segment2(pts, e1, e2)

kwcoco.demo.boids.**_pygame_render_boids**()

Fast and responsive BOID rendering. This is an easter egg.

Requirements: pip install pygame

CommandLine: python -m kwcoco.demo.boids pip install pygame kwcoco -U && python -m kwcoco.demo.boids

kwcoco.demo.boids._yeah_boid()

kwcoco.demo.perterb

Module Contents

Functions

perterb_coco(coco_dset, **kwargs)		Perterbs a coco dataset
_demo_construct_probs(pred_cxs, classes, rng, hacked=1)	pred_scores,	Constructs random probabilities for demo data

kwcoco.demo.perterb.**perterb_coco**(*coco_dset*, ***kwargs*)

Perterbs a coco dataset

Parameters

- **rng** (*int, default=0*)
- **box_noise** (*int, default=0*)
- **cls_noise** (*int, default=0*)
- **null_pred** (*bool, default=False*)
- **with_probs** (*bool, default=False*)
- **score_noise** (*float, default=0.2*)
- **hacked** (*int, default=1*)

Example

```
>>> from kwcoco.demo.perterb import *  # NOQA
>>> from kwcoco.demo.perterb import _demo_construct_probs
>>> import kwcoco
>>> coco_dset = true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': 3,
>>>     'with_probs': 1,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> pred_dset._check_json_serializable()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> gid = 1
>>> canvas = true_dset.delayed_load(gid).finalize()
>>> canvas = true_dset.annots(gid=gid).detections.draw_on(canvas, color='green')
>>> canvas = pred_dset.annots(gid=gid).detections.draw_on(canvas, color='blue')
>>> kwplot.imshow(canvas)
```

Ignore: import xdev from kwcoco.demo.perterb import perterb_coco # NOQA defaultkw = xdev.get_func_kwargs(perterb_coco) for k, v in defaultkw.items():

desc = '' print('{} ({}, default={}): {}'.format(k, type(v).__name__, v, desc))

kwcoco.demo.perterb.**_demo_construct_probs**(*pred_cxs*, *pred_scores*, *classes*, *rng*, *hacked=1*)

Constructs random probabilities for demo data

Example

```
>>> import kwcoco
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> classes = kwcoco.CategoryTree.coerce(10)
>>> hacked = 1
>>> pred_cxs = rng.randint(0, 10, 10)
>>> pred_scores = rng.rand(10)
>>> probs = _demo_construct_probs(pred_cxs, pred_scores, classes, rng, hacked)
>>> probs.sum(axis=1)
```

kwcoco.demo.toydata

Notes

The implementation of *demodata_toy_img* and *demodata_toy_dset* should be redone using the tools built for *random_video_dset*, which have more extensible implementations.

Module Contents

Functions

demodata_toy_dset(image_size=(600, 600), n_imgs=5, verbose=3, rng=0, newstyle=True, dpath=None, bundle_dpath=None, aux=None, use_cache=True, **kwargs)	Create a toy detection problem
random_video_dset(num_videos=1, num_frames=2, num_tracks=2, anchors=None, image_size=(600, 600), verbose=3, render=False, aux=None, multispectral=False, rng=None, dpath=None, **kwargs)	Create a toy Coco Video Dataset
random_single_video_dset(image_size=(600, 600), num_frames=5, num_tracks=3, tid_start=1, gid_start=1, video_id=1, anchors=None, rng=None, render=False, dpath=None, autobuild=True, verbose=3, aux=None, multispectral=False, **kwargs)	Create the video scene layout of object positions.
demodata_toy_img(anchors=None, image_size=(104, 104), categories=None, n_annots=(0, 50), fg_scale=0.5, bg_scale=0.8, bg_intensity=0.1, fg_intensity=0.9, gray=True, centerobj=None, exact=False, newstyle=True, rng=None, aux=None, **kwargs)	Generate a single image with non-overlapping toy objects of available

continues on next page

Table 36 – continued from previous page

render_toy_dataset(dset, rng, dpath=None, renderkw=None)	Create toydata renderings for a preconstructed coco dataset.
render_toy_image(dset, gid, rng=None, renderkw=None)	Modifies dataset inplace, rendering synthetic annotations.
false_color(twochan)	
render_background(img, rng, gray, bg_intensity, bg_scale)	
random_multi_object_path(num_objects, num_frames, rng=None)	import kwarray
random_path(num, degree=1, dimension=2, rng=None, mode='boid')	Create a random path using a somem ethod curve.

Attributes

profile
TOYDATA_VERSION

kwcoco.demo.toydata.**profile**

kwcoco.demo.toydata.**TOYDATA_VERSION = 16**

kwcoco.demo.toydata.**demodata_toy_dset**(*image_size=(600, 600)*, *n_imgs=5*, *verbose=3*, *rng=0*, *newstyle=True*, *dpath=None*, *bundle_dpath=None*, *aux=None*, *use_cache=True*, ***kwargs*)

Create a toy detection problem

Parameters

- **image_size** (*Tuple[int, int]*) – The width and height of the generated images
- **n_imgs** (*int*) – number of images to generate
- **rng** (*int | RandomState, default=0*) – random number generator or seed
- **newstyle** (*bool, default=True*) – create newstyle kwcoco data
- **dpath** (*str*) – path to the directory that will contain the bundle, (defaults to a kwcoco cache dir). Ignored if *bundle_dpath* is given.
- **bundle_dpath** (*str*) – path to the directory that will store images. If specified, dpath is ignored. If unspecified, a bundle will be written inside *dpath*.
- **aux** (*bool*) – if True generates dummy auxiliary channels
- **verbose** (*int, default=3*) – verbosity mode
- **use_cache** (*bool, default=True*) – if True caches the generated json in the *dpath*.
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

Returns

Return type *kwcoco.CocoDataset*

SeeAlso: random_video_dset

CommandLine: xdoctest -m kwcoco.demo.toydata demodata_toy_dset –show

Ignore: import xdev globals().update(xdev.get_func_kwargs(demodata_toy_dset))

Todo:

- [ ] Non-homogeneous images sizes

Example

```
>>> from kwcoco.demo.toydata import *
>>> import kwcoco
>>> dset = demodata_toy_dset(image_size=(300, 300), aux=True, use_cache=False)
>>> # xdoctest: +REQUIRES(--show)
>>> print(ub.repr2(dset.dataset, nl=2))
>>> import kwplot
>>> kwplot.autompl()
>>> dset.show_image(gid=1)
>>> ub.startfile(dset.bundle_dpath)
```

dset._tree()

```
>>> from kwcoco.demo.toydata import *
>>> import kwcoco
```

dset = demodata_toy_dset(image_size=(300, 300), aux=True, use_cache=False) print(dset.imgs[1]) dset._tree()

dset = demodata_toy_dset(image_size=(300, 300), aux=True, use_cache=False, bundle_dpath='test_bundle')

print(dset.imgs[1]) dset._tree()

dset = demodata_toy_dset(image_size=(300, 300), aux=True, use_cache=False, dpath='test_cache_dpath')

kwcoco.demo.toydata.**random_video_dset**(*num_videos=1*, *num_frames=2*, *num_tracks=2*, *anchors=None*, *image_size=(600, 600)*, *verbose=3*, *render=False*, *aux=None*, *multispectral=False*, *rng=None*, *dpath=None*, ***kwargs*)

Create a toy Coco Video Dataset

Parameters

- **num_videos** (*int*) – number of videos
- **num_frames** (*int*) – number of images per video
- **num_tracks** (*int*) – number of tracks per video
- **image_size** (*Tuple[int, int]*) – The width and height of the generated images
- **render** (*bool | dict*) – if truthy the toy annotations are synthetically rendered. See *render_toy_image()* for details.
- **rng** (*int | None | RandomState*) – random seed / state
- **dpath** (*str*) – only used if render is truthy, place to write rendered images.
- **verbose** (*int, default=3*) – verbosity mode
- **aux** (*bool*) – if True generates dummy auxiliary channels

- **multispectral** (*bool*) – similar to aux, but does not have the concept of a "main" image.
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

SeeAlso: random_single_video_dset

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> dset = random_video_dset(render=True, num_videos=3, num_frames=2,
>>>                          num_tracks=10)
>>> # xdoctest: +REQUIRES(--show)
>>> dset.show_image(1, doclf=True)
>>> dset.show_image(2, doclf=True)
```

```
>>> from kwcoco.demo.toydata import *  # NOQA
dset = random_video_dset(render=False, num_videos=3, num_frames=2,
    num_tracks=10)
dset._tree()
dset.imgs[1]
```

dset = random_single_video_dset() dset._tree() dset.imgs[1]

from kwcoco.demo.toydata import * # NOQA dset = random_video_dset(render=True, num_videos=3, num_frames=2,

num_tracks=10)

print(dset.imgs[1]) print('dset.bundle_dpath = {!r}'.format(dset.bundle_dpath)) dset._tree()

import xdev globals().update(xdev.get_func_kwargs(random_video_dset)) num_videos = 2

kwcoco.demo.toydata.**random_single_video_dset**(*image_size=(600, 600)*, *num_frames=5*, *num_tracks=3*, *tid_start=1*, *gid_start=1*, *video_id=1*, *anchors=None*, *rng=None*, *render=False*, *dpath=None*, *autobuild=True*, *verbose=3*, *aux=None*, *multispectral=False*, ***kwargs*)

Create the video scene layout of object positions.

Parameters

- **image_size** (*Tuple[int, int]*) – size of the images
- **num_frames** (*int*) – number of frames in this video
- **num_tracks** (*int*) – number of tracks in this video
- **tid_start** (*int, default=1*) – track-id start index
- **gid_start** (*int, default=1*) – image-id start index
- **video_id** (*int, default=1*) – video-id of this video
- **anchors** (*ndarray | None*) – base anchor sizes of the object boxes we will generate.
- **rng** (*RandomState*) – random state / seed
- **render** (*bool | dict*) – if truthy, does the rendering according to provided params in the case of dict input.
- **autobuild** (*bool, default=True*) – prebuild coco lookup indexes
- **verbose** (*int*) – verbosity level

- **aux** (*bool | List[str]*) – if specified generates auxiliary channels
- **multispectral** (*bool*) – if specified simulates multispectral imagry This is similar to aux, but has no "main" file.
- ****kwargs** – used for old backwards compatible argument names gsize - alias for image_size

Todo:

- [ ] Need maximum allowed object overlap measure
- [ ] Need better parameterized path generation

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> anchors = np.array([ [0.3, 0.3],  [0.1, 0.1]])
>>> dset = random_single_video_dset(render=True, num_frames=10, num_tracks=10,␣
↪anchors=anchors)
>>> # xdoctest: +REQUIRES(--show)
>>> # Show the tracks in a single image
>>> import kwplot
>>> kwplot.autompl()
>>> annots = dset.annots()
>>> tids = annots.lookup('track_id')
>>> tid_to_aids = ub.group_items(annots.aids, tids)
>>> paths = []
>>> track_boxes = []
>>> for tid, aids in tid_to_aids.items():
>>>     boxes = dset.annots(aids).boxes.to_cxywh()
>>>     path = boxes.data[:, 0:2]
>>>     paths.append(path)
>>>     track_boxes.append(boxes)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> ax = plt.gca()
>>> ax.cla()
>>> #
>>> import kwimage
>>> colors = kwimage.Color.distinct(len(track_boxes))
>>> for i, boxes in enumerate(track_boxes):
>>>     color = colors[i]
>>>     path = boxes.data[:, 0:2]
>>>     boxes.draw(color=color, centers={'radius': 0.01}, alpha=0.5)
>>>     ax.plot(path.T[0], path.T[1], 'x-', color=color)
```

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> anchors = np.array([ [0.2, 0.2],  [0.1, 0.1]])
>>> gsize = np.array([(600, 600)])
>>> print(anchors * gsize)
>>> dset = random_single_video_dset(render=True, num_frames=10,
>>>                                 anchors=anchors, num_tracks=10)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> plt.clf()
>>> gids = list(dset.imgs.keys())
>>> pnums = kwplot.PlotNums(nSubplots=len(gids), nRows=1)
>>> for gid in gids:
>>>     dset.show_image(gid, pnum=pnums(), fnum=1, title=False)
>>> pnums = kwplot.PlotNums(nSubplots=len(gids))
```

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> dset = random_single_video_dset(num_frames=10, num_tracks=10, aux=True)
>>> assert 'auxiliary' in dset.imgs[1]
>>> assert dset.imgs[1]['auxiliary'][0]['channels']
>>> assert dset.imgs[1]['auxiliary'][1]['channels']
```

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> multispectral = True
>>> dset = random_single_video_dset(num_frames=1, num_tracks=1, multispectral=True)
>>> dset._check_json_serializable()
>>> dset.dataset['images']
>>> assert dset.imgs[1]['auxiliary'][1]['channels']
>>> # test that we can render
>>> render_toy_dataset(dset, rng=0, dpath=None, renderkw={})
```

Ignore: import xdev globals().update(xdev.get_func_kwargs(random_single_video_dset))

Ignore:

dset = random_single_video_dset(render=True, num_frames=10, anchors=anchors, num_tracks=10)

dset._tree()

kwcoco.demo.toydata.**demodata_toy_img**(*anchors=None*, *image_size=(104, 104)*, *categories=None*, *n_annots=(0, 50)*, *fg_scale=0.5*, *bg_scale=0.8*, *bg_intensity=0.1*, *fg_intensity=0.9*, *gray=True*, *centerobj=None*, *exact=False*, *newstyle=True*, *rng=None*, *aux=None*, ***kwargs*)

Generate a single image with non-overlapping toy objects of available categories.

Todo:

DEPRECATE IN FAVOR OF random_single_video_dset + render_toy_image

Parameters

- **anchors** (*ndarray*) – Nx2 base width / height of boxes
- **gsize** (*Tuple[int, int]*) – width / height of the image
- **categories** (*List[str]*) – list of category names
- **n_annots** (*Tuple | int*) – controls how many annotations are in the image. if it is a tuple, then it is interpreted as uniform random bounds
- **fg_scale** (*float*) – standard deviation of foreground intensity
- **bg_scale** (*float*) – standard deviation of background intensity
- **bg_intensity** (*float*) – mean of background intensity
- **fg_intensity** (*float*) – mean of foreground intensity
- **centerobj** (*bool*) – if 'pos', then the first annotation will be in the center of the image, if 'neg', then no annotations will be in the center.
- **exact** (*bool*) – if True, ensures that exactly the number of specified annots are generated.
- **newstyle** (*bool*) – use new-sytle kwcoco format
- **rng** (*RandomState*) – the random state used to seed the process
- **aux** – if specified builds auxiliary channels
- ****kwargs** – used for old backwards compatible argument names. gsize - alias for image_size

CommandLine: xdoctest -m kwcoco.demo.toydata demodata_toy_img:0 –profile xdoctest -m kwcoco.demo.toydata demodata_toy_img:1 –show

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> img, anns = demodata_toy_img(image_size=(32, 32), anchors=[[.3, .3]], rng=0)
>>> img['imdata'] = '<ndarray shape={}>'.format(img['imdata'].shape)
>>> print('img = {}'.format(ub.repr2(img)))
>>> print('anns = {}'.format(ub.repr2(anns, nl=2, cbr=True)))
>>> # xdoctest: +IGNORE_WANT
img = {
    'height': 32,
    'imdata': '<ndarray shape=(32, 32, 3)>',
    'width': 32,
}
anns = [{'bbox': [15, 10, 9, 8],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': '[`06j0000O20N1000e8', 'size': [32, 32]},},
 {'bbox': [11, 20, 7, 7],
  'category_name': 'star',
```

(continues on next page)

(continued from previous page)

```
  'keypoints': [],
  'segmentation': {'counts': 'g;1m04N0O20N102L[=', 'size': [32, 32]},},
 {'bbox': [4, 4, 8, 6],
  'category_name': 'superstar',
  'keypoints': [{'keypoint_category': 'left_eye', 'xy': [7.25, 6.8125]}, {'keypoint_
↪category': 'right_eye', 'xy': [8.75, 6.8125]}],
  'segmentation': {'counts': 'U4210j0300O01010O00MVO0ed0', 'size': [32, 32]},},
 {'bbox': [3, 20, 6, 7],
  'category_name': 'star',
  'keypoints': [],
  'segmentation': {'counts': 'g31m04N000002L[f0', 'size': [32, 32]},},]
```

Example

```
>>> # xdoctest: +REQUIRES(--show)
>>> img, anns = demodata_toy_img(image_size=(172, 172), rng=None, aux=True)
>>> print('anns = {}'.format(ub.repr2(anns, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'], pnum=(1, 2, 1), fnum=1)
>>> auxdata = img['auxiliary'][0]['imdata']
>>> kwplot.imshow(auxdata, pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```

Example

```
>>> # xdoctest: +REQUIRES(--show)
>>> img, anns = demodata_toy_img(image_size=(172, 172), rng=None, aux=True)
>>> print('anns = {}'.format(ub.repr2(anns, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'], pnum=(1, 2, 1), fnum=1)
>>> auxdata = img['auxiliary'][0]['imdata']
>>> kwplot.imshow(auxdata, pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```

Ignore: from kwcoco.demo.toydata import * import xinspect globals().update(xinspect.get_kwargs(demodata_toy_img))

kwcoco.demo.toydata.**render_toy_dataset**(*dset*, *rng*, *dpath=None*, *renderkw=None*)

Create toydata renderings for a preconstructed coco dataset.

Parameters

- **dset** (*CocoDataset*) – A dataset that contains special “renderable” annotations. (e.g. the demo shapes). Each image can contain special fields that influence how an image will be rendered.

  Currently this process is simple, it just creates a noisy image with the shapes superimposed over where they should exist as indicated by the annotations. In the future this may become more sophisticated.

Each item in *dset.dataset['images']* will be modified to add the "file_name" field indicating where the rendered data is writen.

- **rng** (*int | None | RandomState*) – random state
- **dpath** (*str*) – The location to write the images to. If unspecified, it is written to the rendered folder inside the kwcoco cache directory.
- **renderkw** (*dict*) – See *render_toy_image()* for details.

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> import kwarray
>>> rng = None
>>> rng = kwarray.ensure_rng(rng)
>>> num_tracks = 3
>>> dset = random_video_dset(rng=rng, num_videos=3, num_frames=10, num_tracks=3)
>>> dset = render_toy_dataset(dset, rng)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> plt.clf()
>>> gids = list(dset.imgs.keys())
>>> pnums = kwplot.PlotNums(nSubplots=len(gids), nRows=num_tracks)
>>> for gid in gids:
>>>     dset.show_image(gid, pnum=pnums(), fnum=1, title=False)
>>> pnums = kwplot.PlotNums(nSubplots=len(gids))
>>> #
>>> # for gid in gids:
>>> #     canvas = dset.draw_image(gid)
>>> #     kwplot.imshow(canvas, pnum=pnums(), fnum=2)
```

kwcoco.demo.toydata.**render_toy_image**(*dset*, *gid*, *rng=None*, *renderkw=None*)

Modifies dataset inplace, rendering synthetic annotations.

This does not write to disk. Instead this writes to placeholder values in the image dictionary.

Parameters

- **dset** (*CocoDataset*) – coco dataset with renderable anotations / images
- **gid** (*int*) – image to render
- **rng** (*int | None | RandomState*) – random state
- **renderkw** (*dict*) – rendering config gray (boo): gray or color images fg_scale (float): foreground noisyness (gauss std) bg_scale (float): background noisyness (gauss std) fg_intensity (float): foreground brightness (gauss mean) bg_intensity (float): background brightness (gauss mean) newstyle (bool): use new kwcoco datastructure formats with_kpts (bool): include keypoint info with_sseg (bool): include segmentation info

Returns the inplace-modified image dictionary

Return type Dict

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> image_size=(600, 600)
>>> num_frames=5
>>> verbose=3
>>> rng = None
>>> import kwarray
>>> rng = kwarray.ensure_rng(rng)
>>> aux = 'mx'
>>> dset = random_single_video_dset(
>>>     image_size=image_size, num_frames=num_frames, verbose=verbose, aux=aux,␣
↪rng=rng)
>>> print('dset.dataset = {}'.format(ub.repr2(dset.dataset, nl=2)))
>>> gid = 1
>>> renderkw = {}
>>> render_toy_image(dset, gid, rng, renderkw=renderkw)
>>> img = dset.imgs[gid]
>>> canvas = img['imdata']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(canvas, doclf=True, pnum=(1, 2, 1))
>>> dets = dset.annots(gid=gid).detections
>>> dets.draw()
```

```
>>> auxdata = img['auxiliary'][0]['imdata']
>>> aux_canvas = false_color(auxdata)
>>> kwplot.imshow(aux_canvas, pnum=(1, 2, 2))
>>> _ = dets.draw()
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> img, anns = demodata_toy_img(image_size=(172, 172), rng=None, aux=True)
>>> print('anns = {}'.format(ub.repr2(anns, nl=1)))
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'], pnum=(1, 2, 1), fnum=1)
>>> auxdata = img['auxiliary'][0]['imdata']
>>> kwplot.imshow(auxdata, pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> multispectral = True
>>> dset = random_single_video_dset(num_frames=1, num_tracks=1, multispectral=True)
>>> gid = 1
>>> dset.imgs[gid]
>>> rng = kwarray.ensure_rng(0)
>>> renderkw = {'with_sseg': True}
>>> img = render_toy_image(dset, gid, rng=rng, renderkw=renderkw)
```

kwcoco.demo.toydata.**false_color**(*twochan*)

kwcoco.demo.toydata.**render_background**(*img*, *rng*, *gray*, *bg_intensity*, *bg_scale*)

kwcoco.demo.toydata.**random_multi_object_path**(*num_objects*, *num_frames*, *rng=None*)
import kwarray import kwplot plt = kwplot.autoplt()

num_objects = 5 num_frames = 100 rng = kwarray.ensure_rng(0)

from kwcoco.demo.toydata import * # NOQA paths = random_multi_object_path(num_objects, num_frames, rng)

from mpl_toolkits.mplot3d import Axes3D # NOQA ax = plt.gca(projection='3d') ax.cla()

for path in paths: time = np.arange(len(path)) ax.plot(time, path.T[0] * 1, path.T[1] * 1, 'o-')

ax.set_xlim(0, num_frames) ax.set_ylim(-.01, 1.01) ax.set_zlim(-.01, 1.01)

kwcoco.demo.toydata.**random_path**(*num*, *degree=1*, *dimension=2*, *rng=None*, *mode='boid'*)
Create a random path using a somem ethod curve.

Parameters

- **num** (*int*) – number of points in the path
- **degree** (*int, default=1*) – degree of curvieness of the path
- **dimension** (*int, default=2*) – number of spatial dimensions
- **mode** (*str*) – can be boid, walk, or bezier
- **rng** (*RandomState, default=None*) – seed

References

https://github.com/dhermes/bezier

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> num = 10
>>> dimension = 2
>>> degree = 3
>>> rng = None
>>> path = random_path(num, degree, dimension, rng, mode='boid')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> kwplot.multi_plot(xdata=path[:, 0], ydata=path[:, 1], fnum=1, doclf=1, xlim=(0,␣
↪1), ylim=(0, 1))
>>> kwplot.show_if_requested()
```

Example

```
>>> # xdoctest: +REQUIRES(--3d)
>>> # xdoctest: +REQUIRES(module:bezier)
>>> import kwarray
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> #
>>> num= num_frames = 100
>>> rng = kwarray.ensure_rng(0)
>>> #
>>> from kwcoco.demo.toydata import *  # NOQA
>>> paths = []
>>> paths.append(random_path(num, degree=3, dimension=3, mode='bezier'))
>>> paths.append(random_path(num, degree=2, dimension=3, mode='bezier'))
>>> paths.append(random_path(num, degree=4, dimension=3, mode='bezier'))
>>> #
>>> from mpl_toolkits.mplot3d import Axes3D  # NOQA
>>> ax = plt.gca(projection='3d')
>>> ax.cla()
>>> #
>>> for path in paths:
>>>     time = np.arange(len(path))
>>>     ax.plot(time, path.T[0] * 1, path.T[1] * 1, 'o-')
>>> ax.set_xlim(0, num_frames)
>>> ax.set_ylim(-.01, 1.01)
>>> ax.set_zlim(-.01, 1.01)
>>> ax.set_xlabel('x')
>>> ax.set_ylabel('y')
>>> ax.set_zlabel('z')
```

kwcoco.demo.toypatterns

Module Contents

Classes

CategoryPatterns

Example

Rasters

Functions

star(a, dtype=np.uint8)	Generates a star shaped structuring element.

class kwcoco.demo.toypatterns.**CategoryPatterns**(*categories=None*, *fg_scale=0.5*, *fg_intensity=0.9*, *rng=None*)

Bases: object

Example

```
>>> self = CategoryPatterns.coerce()
>>> chip = np.zeros((100, 100, 3))
>>> offset = (20, 10)
>>> dims = (160, 140)
>>> info = self.random_category(chip, offset, dims)
>>> print('info = {}'.format(ub.repr2(info, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(info['data'], pnum=(1, 2, 1), fnum=1, title='chip-space')
>>> kpts = kwimage.Points._from_coco(info['keypoints'])
>>> kpts.translate(-np.array(offset)).draw(radius=3)
>>> #####
>>> mask = kwimage.Mask.coerce(info['segmentation'])
>>> kwplot.imshow(mask.to_c_mask().data, pnum=(1, 2, 2), fnum=1, title='img-space')
>>> kpts.draw(radius=3)
>>> kwplot.show_if_requested()
```

_default_categories

_default_keypoint_categories

_default_catnames = ['star', 'eff', 'superstar']

classmethod coerce(*CategoryPatterns*, *data=None*, ***kwargs*)

Construct category patterns from either defaults or only with specific categories. Can accept either an existig category pattern object, a list of known catnames, or mscoco category dictionaries.

Example

```
>>> data = ['superstar']
>>> self = CategoryPatterns.coerce(data)
```

__len__(*self*)

__getitem__(*self*, *index*)

__iter__(*self*)

index(*self*, *name*)

get(*self*, *index*, *default=ub.NoParam*)

random_category(*self*, *chip*, *xy_offset=None*, *dims=None*, *newstyle=True*, *size=None*)

Ignore: import xdev globals().update(xdev.get_func_kwargs(self.random_category))

Example

```
>>> from kwcoco.demo.toypatterns import *  # NOQA
>>> self = CategoryPatterns.coerce(['superstar'])
>>> chip = np.random.rand(64, 64)
>>> info = self.random_category(chip)
```

render_category(*self*, *cname*, *chip*, *xy_offset=None*, *dims=None*, *newstyle=True*, *size=None*)

Ignore: import xdev globals().update(xdev.get_func_kwargs(self.random_category))

Example

```
>>> from kwcoco.demo.toypatterns import *  # NOQA
>>> self = CategoryPatterns.coerce(['superstar'])
>>> chip = np.random.rand(64, 64)
>>> info = self.render_category('superstar', chip, newstyle=True)
>>> print('info = {}'.format(ub.repr2(info, nl=-1)))
>>> info = self.render_category('superstar', chip, newstyle=False)
>>> print('info = {}'.format(ub.repr2(info, nl=-1)))
```

Example

```
>>> from kwcoco.demo.toypatterns import *  # NOQA
>>> self = CategoryPatterns.coerce(['superstar'])
>>> chip = None
>>> dims = (64, 64)
>>> info = self.render_category('superstar', chip, newstyle=True, dims=dims,␣
↪size=dims)
>>> print('info = {}'.format(ub.repr2(info, nl=-1)))
```

_todo_refactor_geometric_info(*self*, *cname*, *xy_offset*, *dims*)

This function is used to populate kpts and sseg information in the autogenerated coco dataset before rendering. It is redundant with other functionality.

TODO: rectify with _from_elem

Example

```
>>> self = CategoryPatterns.coerce(['superstar'])
>>> dims = (64, 64)
>>> cname = 'superstar'
>>> xy_offset = None
>>> self._todo_refactor_geometric_info(cname, xy_offset, dims)
```

_package_info(*self*, *cname*, *data*, *mask*, *kpts*, *xy_offset*, *dims*, *newstyle*)

packages data from _from_elem into coco-like annotation

_from_elem(*self*, *cname*, *chip*, *size=None*)

Example

```
>>> # hack to allow chip to be None
>>> chip = None
>>> size = (32, 32)
>>> cname = 'superstar'
>>> self = CategoryPatterns.coerce()
>>> self._from_elem(cname, chip, size)
```

kwcoco.demo.toypatterns.**star**(*a*, *dtype=np.uint8*)
Generates a star shaped structuring element.

Much faster than skimage.morphology version

class kwcoco.demo.toypatterns.**Rasters**

static superstar()
test data patch

Ignore:

```
>>> kwplot.autompl()
>>> patch = Rasters.superstar()
>>> data = np.clip(kwimage.imscale(patch, 2.2), 0, 1)
>>> kwplot.imshow(data)
```

static eff()
test data patch

Ignore:

```
>>> kwplot.autompl()
>>> eff = kwimage.draw_text_on_image(None, 'F', (0, 1), valign='top')
>>> patch = Rasters.eff()
>>> data = np.clip(kwimage.imscale(Rasters.eff(), 2.2), 0, 1)
>>> kwplot.imshow(data)
```

2.2.4 kwcoco.examples

Submodules

kwcoco.examples.draw_gt_and_predicted_boxes

Module Contents

Functions

draw_true_and_pred_boxes(true_fpath, pred_fpath, gid, viz_fpath)	How do you generally visualize gt and predicted bounding boxes together?

kwcoco.examples.draw_gt_and_predicted_boxes.**draw_true_and_pred_boxes**(*true_fpath*, *pred_fpath*, *gid*, *viz_fpath*)

How do you generally visualize gt and predicted bounding boxes together?

Example

```
>>> import kwcoco
>>> import ubelt as ub
>>> from os.path import join
>>> from kwcoco.demo.perterb import perterb_coco
>>> # Create a working directory
>>> dpath = ub.ensure_app_cache_dir('kwcoco/examples/draw_true_and_pred_boxes')
>>> # Lets setup some dummy true data
>>> true_dset = kwcoco.CocoDataset.demo('shapes2')
>>> true_dset.fpath = join(dpath, 'true_dset.kwcoco.json')
>>> true_dset.dump(true_dset.fpath, newlines=True)
>>> # Lets setup some dummy predicted data
>>> pred_dset = perterb_coco(true_dset, box_noise=100, rng=421)
>>> pred_dset.fpath = join(dpath, 'pred_dset.kwcoco.json')
>>> pred_dset.dump(pred_dset.fpath, newlines=True)
>>> #
>>> # We now have our true and predicted data, lets visualize
>>> true_fpath = true_dset.fpath
>>> pred_fpath = pred_dset.fpath
>>> print('dpath = {!r}'.format(dpath))
>>> print('true_fpath = {!r}'.format(true_fpath))
>>> print('pred_fpath = {!r}'.format(pred_fpath))
>>> # Lets choose an image id to visualize and a path to write to
>>> gid = 1
>>> viz_fpath = join(dpath, 'viz_{}.jpg'.format(gid))
>>> # The answer to the question is in the logic of this function
>>> draw_true_and_pred_boxes(true_fpath, pred_fpath, gid, viz_fpath)
```

kwcoco.examples.getting_started_existing_dataset

Module Contents

Functions

getting_started_existing_dataset()	If you want to start using the Python API. Just open IPython and try:
the_core_dataset_backend()	

continues on next page

Table 41 – continued from previous page

demo_vectorize_interface()	>>> import kwcoco

kwcoco.examples.getting_started_existing_dataset.**getting_started_existing_dataset**()
If you want to start using the Python API. Just open IPython and try:

kwcoco.examples.getting_started_existing_dataset.**the_core_dataset_backend**()

kwcoco.examples.getting_started_existing_dataset.**demo_vectorize_interface**()

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes2')
>>> #
>>> aids = [1, 2, 3, 4]
>>> annots = dset.annots(aids)
...
>>> print('annots = {!r}'.format(annots))
annots = <Annots(num=4) at ...>
```

```
>>> annots.lookup('bbox')
[[346.5, 335.2, 33.2, 99.4],
 [344.5, 327.7, 48.8, 111.1],
 [548.0, 154.4, 57.2, 62.1],
 [548.7, 151.0, 59.4, 80.5]]
```

```
>>> gids = annots.lookup('image_id')
>>> print('gids = {!r}'.format(gids))
gids = [1, 2, 1, 2]
```

```
>>> images = dset.images(gids)
>>> list(zip(images.lookup('width'), images.lookup('height')))
[(600, 600), (600, 600), (600, 600), (600, 600)]
```

kwcoco.examples.modification_example

Module Contents

Functions

dataset_modification_example_via_copy()	Say you are given a dataset as input and you need to add your own
dataset_modification_example_via_construction()	Alternatively you can make a new dataset and copy over categories / images

kwcoco.examples.modification_example.**dataset_modification_example_via_copy**()
Say you are given a dataset as input and you need to add your own annotation "predictions" to it. You could copy the existing dataset, remove all the annotations, and then add your new annotations.

kwcoco.examples.modification_example.**dataset_modification_example_via_construction**()
Alternatively you can make a new dataset and copy over categories / images as needed

kwcoco.examples.simple_kwcoco_torch_dataset

This example demonstrates how to use kwcoco to write a very simple torch dataset. This assumes the dataset will be single-image RGB inputs. This file is intended to talk the reader through what we are doing and why.

This example aims for clairity over being concise. There are APIs exposed by kwcoco (and its sister module ndsampler) that can perform the same tasks more efficiently and with fewer lines of code.

If you run the doctest, it will produce a visualization that shows the images with boxes drawn on it, running it multiple times will let you see the augmentations. This can be done with the following command:

xdoctest -m kwcoco.examples.simple_kwcoco_torch_dataset KWCocoSimpleTorchDataset –show

Or just copy the doctest into IPython and run it.

Module Contents

Classes

KWCocoSimpleTorchDataset	A simple torch dataloader where each image is considered a single item.

Attributes

DatasetBase

kwcoco.examples.simple_kwcoco_torch_dataset.**DatasetBase**

class kwcoco.examples.simple_kwcoco_torch_dataset.**KWCocoSimpleTorchDataset**(*coco_dset*, *input_dims=None*, *antialias=False*, *rng=None*)

Bases: *DatasetBase*

A simple torch dataloader where each image is considered a single item.

Parameters

- **coco_dset** (*kwcoco.CocoDataset | str*) – something coercable to a kwcoco dataset, this could either be a *kwcoco.CocoDataset* object, a path to a kwcoco manifest on disk, or a special toydata code. See `kwcoco.CocoDataset.coerce()` for more details.
- **input_dims** (*Tuple[int, int]*) – These are the (height, width) dimensions that the image will be resized to.
- **antialias** (*bool, default=False*) – If true, we will antialias before downsampling.
- **rng** (*RandomState | int | None*) – an existing random number generator or a random seed to produce deterministic augmentations.

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> from kwcoco.examples.simple_kwcoco_torch_dataset import *  # NOQA
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.demo('shapes8')
>>> input_dims = (384, 384)
>>> self = torch_dset = KWCocoSimpleTorchDataset(coco_dset, input_dims=input_dims)
>>> index = len(self) // 2
>>> item = self[index]
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.figure(doclf=True, fnum=1)
>>> kwplot.autompl()
>>> canvas = item['inputs']['rgb'].numpy().transpose(1, 2, 0)
>>> # Construct kwimage objects for batch item visualization
>>> dets = kwimage.Detections(
>>>     boxes=kwimage.Boxes(item['labels']['cxywh'], 'cxywh'),
>>>     class_idxs=item['labels']['class_idxs'],
>>>     classes=self.classes,
>>> ).numpy()
>>> # Overlay annotations on the image
>>> canvas = dets.draw_on(canvas)
>>> kwplot.imshow(canvas)
>>> kwplot.show_if_requested()
```

__len__(*self*)
Return the number of items in the Dataset

__getitem__(*self*, *index*)
Construct a batch item to be used in training

2.2.5 kwcoco.metrics

mkinit kwcoco.metrics -w –relative

Submodules

kwcoco.metrics.assignment

---

when deciding whether to go up or down in the tree.

[ ] medschool applications true-pred matching (applicant proposing) fast

algorithm.

[ ] Maybe looping over truth rather than pred is faster? but it makes you

have to combine pred score / ious, which is weird.

[x] preallocate ndarray and use hstack to build confusion vectors?

doesn't help

[ ] relevant classes / classes / classes-of-interest we care about needs

to be a first class member of detection metrics.:

Module Contents

Functions

_assign_confusion_vectors(true_dets, pred_dets, bg_weight=1.0, iou_thresh=0.5, bg_cidx=-1, bias=0.0, classes=None, compat='all', prioritize='iou', ignore_classes='ignore', max_dets=None)	Create confusion vectors for detections by assigning to ground true boxes
_critical_loop(true_dets, pred_dets, iou_lookup, isvalid_lookup, cx_to_matchable_txs, bg_weight, prioritize, iou_thresh_, pdist_priority, cx_to_ancestors, bg_cidx, ignore_classes, max_dets)	
_fast_pdist_priority(classes, prioritize, _cache={})	Custom priority computation. Needs some vetting.
_filter_ignore_regions(true_dets, pred_dets, ioaa_thresh=0.5, ignore_classes='ignore')	Determine which true and predicted detections should be ignored.

Attributes

USE_NEG_INF

kwcoco.metrics.assignment.**USE_NEG_INF = True**

kwcoco.metrics.assignment.**_assign_confusion_vectors**(*true_dets*, *pred_dets*, *bg_weight=1.0*, *iou_thresh=0.5*, *bg_cidx=- 1*, *bias=0.0*, *classes=None*, *compat='all'*, *prioritize='iou'*, *ignore_classes='ignore'*, *max_dets=None*)

Create confusion vectors for detections by assigning to ground true boxes

Given predictions and truth for an image return (y_pred, y_true, y_score), which is suitable for sklearn classification metrics

Parameters

- **true_dets** (*Detections*) – groundtruth with boxes, classes, and weights
- **pred_dets** (*Detections*) – predictions with boxes, classes, and scores
- **iou_thresh** (*float, default=0.5*) – bounding box overlap iou threshold required for assignment
- **bias** (*float, default=0.0*) – for computing bounding box overlap, either 1 or 0
- **gids** (*List[int], default=None*) – which subset of images ids to compute confusion metrics on. If not specified all images are used.
- **compat** (*str, default='all'*) – can be ('ancestors' | 'mutex' | 'all'). determines which pred boxes are allowed to match which true boxes. If 'mutex', then pred boxes can only match true boxes of the same class. If 'ancestors', then pred boxes can match true boxes that match

or have a coarser label. If 'all', then any pred can match any true, regardless of its category label.

- **prioritize** (*str, default='iou'*) – can be ('iou' | 'class' | 'correct') determines which box to assign to if mutiple true boxes overlap a predicted box. if prioritize is iou, then the true box with maximum iou (above iou_thresh) will be chosen. If prioritize is class, then it will prefer matching a compatible class above a higher iou. If prioritize is correct, then ancestors of the true class are preferred over descendents of the true class, over unreleated classes.
- **bg_cidx** (*int, default=-1*) – The index of the background class. The index used in the truth column when a predicted bounding box does not match any true bounding box.
- **classes** (*List[str] | kwcoco.CategoryTree*) – mapping from class indices to class names. Can also contain class heirarchy information.
- **ignore_classes** (*str | List[str]*) – class name(s) indicating ignore regions
- **max_dets** (*int*) – maximum number of detections to consider

---

Todo:

- [ ] This is a bottleneck function. An implementation in C / C++ /

Cython would likely improve the overall system.

- **[ ] Implement crowd truth. Allow multiple predictions to match any** truth objet marked as "iscrowd".

Returns

with relevant confusion vectors. This keys of this dict can be interpreted as columns of a data frame. The *txs* / *pxs* columns represent the indexes of the true / predicted annotations that were assigned as matching. Additionally each row also contains the true and predicted class index, the predicted score, the true weight and the iou of the true and predicted boxes. A *txs* value of -1 means that the predicted box was not assigned to a true annotation and a *pxs* value of -1 means that the true annotation was not assigne to any predicted annotation.

Return type dict

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import pandas as pd
>>> import kwimage
>>> # Given a raw numpy representation construct Detection wrappers
>>> true_dets = kwimage.Detections(
>>>     boxes=kwimage.Boxes(np.array([
>>>         [ 0,  0, 10, 10], [10,  0, 20, 10],
>>>         [10,  0, 20, 10], [20,  0, 30, 10]]), 'tlbr'),
>>>     weights=np.array([1, 0, .9, 1]),
>>>     class_idxs=np.array([0, 0, 1, 2]))
>>> pred_dets = kwimage.Detections(
>>>     boxes=kwimage.Boxes(np.array([
>>>         [6, 2, 20, 10], [3,  2, 9, 7],
>>>         [3,  9, 9, 7],  [3,  2, 9, 7],
>>>         [2,  6, 7, 7],  [20,  0, 30, 10]]), 'tlbr'),
>>>     scores=np.array([.5, .5, .5, .5, .5, .5]),
```

(continues on next page)

(continued from previous page)

```
>>>     class_idxs=np.array([0, 0, 1, 2, 0, 1]))
>>> bg_weight = 1.0
>>> compat = 'all'
>>> iou_thresh = 0.5
>>> bias = 0.0
>>> import kwcoco
>>> classes = kwcoco.CategoryTree.from_mutex(list(range(3)))
>>> bg_cidx = -1
>>> y = _assign_confusion_vectors(true_dets, pred_dets, bias=bias,
>>>                               bg_weight=bg_weight, iou_thresh=iou_thresh,
>>>                               compat=compat)
>>> y = pd.DataFrame(y)
>>> print(y)  # xdoc: +IGNORE_WANT
   pred  true  score  weight     iou  txs  pxs
0     1     2 0.5000  1.0000  1.0000    3    5
1     0    -1 0.5000  1.0000 -1.0000   -1    4
2     2    -1 0.5000  1.0000 -1.0000   -1    3
3     1    -1 0.5000  1.0000 -1.0000   -1    2
4     0    -1 0.5000  1.0000 -1.0000   -1    1
5     0     0 0.5000  0.0000  0.6061    1    0
6    -1     0 0.0000  1.0000 -1.0000    0   -1
7    -1     1 0.0000  0.9000 -1.0000    2   -1
```

Ignore: from xinspect.dynamic_kwargs import get_func_kwargs globals().update(get_func_kwargs(_assign_confusion_vectors))

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import pandas as pd
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(nimgs=1, nclasses=8,
>>>                              nboxes=(0, 20), n_fp=20,
>>>                              box_noise=.2, cls_noise=.3)
>>> classes = dmet.classes
>>> gid = 0
>>> true_dets = dmet.true_detections(gid)
>>> pred_dets = dmet.pred_detections(gid)
>>> y = _assign_confusion_vectors(true_dets, pred_dets,
>>>                               classes=dmet.classes,
>>>                               compat='all', prioritize='class')
>>> y = pd.DataFrame(y)
>>> print(y)  # xdoc: +IGNORE_WANT
>>> y = _assign_confusion_vectors(true_dets, pred_dets,
>>>                               classes=dmet.classes,
>>>                               compat='ancestors', iou_thresh=.5)
>>> y = pd.DataFrame(y)
>>> print(y)  # xdoc: +IGNORE_WANT
```

kwcoco.metrics.assignment.**_critical_loop**(*true_dets*, *pred_dets*, *iou_lookup*, *isvalid_lookup*, *cx_to_matchable_txs*, *bg_weight*, *prioritize*, *iou_thresh_*, *pdist_priority*, *cx_to_ancestors*, *bg_cidx*, *ignore_classes*, *max_dets*)

kwcoco.metrics.assignment.**_fast_pdist_priority**(*classes*, *prioritize*, *_cache={}*)

Custom priority computation. Needs some vetting.

This is the priority used when deciding which prediction to assign to which truth.

Todo:

- **[ ] Look at absolute difference in sibling entropy when deciding** whether to go up or down in the tree.

---

kwcoco.metrics.assignment.**_filter_ignore_regions**(*true_dets*, *pred_dets*, *ioaa_thresh=0.5*, *ignore_classes='ignore'*)

Determine which true and predicted detections should be ignored.

Parameters

- **true_dets** (*Detections*)
- **pred_dets** (*Detections*)
- **ioaa_thresh** (*float*) – intersection over other area thresh for ignoring a region.

Returns

flags indicating which true and predicted detections should be ignored.

Return type Tuple[ndarray, ndarray]

Example

```
>>> from kwcoco.metrics.assignment import *  # NOQA
>>> from kwcoco.metrics.assignment import _filter_ignore_regions
>>> import kwimage
>>> pred_dets = kwimage.Detections.random(classes=['a', 'b', 'c'])
>>> true_dets = kwimage.Detections.random(
>>>     segmentations=True, classes=['a', 'b', 'c', 'ignore'])
>>> ignore_classes = {'ignore', 'b'}
>>> ioaa_thresh = 0.5
>>> print('true_dets = {!r}'.format(true_dets))
>>> print('pred_dets = {!r}'.format(pred_dets))
>>> flags1, flags2 = _filter_ignore_regions(
>>>     true_dets, pred_dets, ioaa_thresh=ioaa_thresh, ignore_classes=ignore_
↪classes)
>>> print('flags1 = {!r}'.format(flags1))
>>> print('flags2 = {!r}'.format(flags2))
```

```
>>> flags3, flags4 = _filter_ignore_regions(
>>>     true_dets, pred_dets, ioaa_thresh=ioaa_thresh,
>>>     ignore_classes={c.upper() for c in ignore_classes})
>>> assert np.all(flags1 == flags3)
>>> assert np.all(flags2 == flags4)
```

kwcoco.metrics.clf_report

Module Contents

Functions

classification_report(y_true, y_pred, target_names=None, sample_weight=None, verbose=False, remove_unsupported=False, log=None, ascii_only=False)	Computes a classification report which is a collection of various metrics
ovr_classification_report(mc_y_true, mc_probs, target_names=None, sample_weight=None, metrics=None, verbose=0, remove_unsupported=False, log=None)	One-vs-rest classification report

Attributes

ASCII_ONLY

kwcoco.metrics.clf_report.**ASCII_ONLY**

kwcoco.metrics.clf_report.**classification_report**(*y_true*, *y_pred*, *target_names=None*, *sample_weight=None*, *verbose=False*, *remove_unsupported=False*, *log=None*, *ascii_only=False*)

Computes a classification report which is a collection of various metrics commonly used to evaulate classification quality. This can handle binary and multiclass settings.

Note that this function does not accept probabilities or scores and must instead act on final decisions. See ovr_classification_report for a probability based report function using a one-vs-rest strategy.

This emulates the bm(cm) Matlab script written by David Powers that is used for computing bookmaker, markedness, and various other scores.

References: https://csem.flinders.edu.au/research/techreps/SIE07001.pdf https://www.mathworks.com/matlabcentral/fileexchange/5648-bm-cm-?requestedDomain=www.mathworks.com Jurman, Riccadonna, Furlanello, (2012). A Comparison of MCC and CEN

Error Measures in MultiClass Prediction

Args: y_true (array): true labels for each item y_pred (array): predicted labels for each item target_names (List): mapping from label to category name sample_weight (ndarray): weight for each item verbose (False): print if True log (callable): print or logging function remove_unsupported (bool, default=False): removes categories that have

no support.

ascii_only (bool, default=False): if True dont use unicode characters. if the environ ASCII_ONLY is present this is forced to True and cannot be undone.

Example:

```
>>> # xdoctest: +IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:sklearn)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> y_true = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3]
>>> y_pred = [1, 2, 1, 3, 1, 2, 2, 3, 2, 2, 3, 3, 2, 3, 3, 3, 1, 3]
>>> target_names = None
>>> sample_weight = None
>>> report = classification_report(y_true, y_pred, verbose=0, ascii_
↪only=1)
>>> print(report['confusion'])
pred  1  2  3  r
real
1     3  1  1   5
2     0  4  1   5
3     1  1  6   8
p     4  6  8  18
>>> print(report['metrics'])
metric    precision  recall    fpr  markedness  bookmaker    mcc ␣
↪support
class
1            0.7500  0.6000 0.0769      0.6071     0.5231 0.5635       ␣
↪5
2            0.6667  0.8000 0.1538      0.5833     0.6462 0.6139       ␣
↪5
3            0.7500  0.7500 0.2000      0.5500     0.5500 0.5500       ␣
↪8
combined     0.7269  0.7222 0.1530      0.5751     0.5761 0.5758      ␣
↪18
```

Example:

```
>>> # xdoctest: +IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:sklearn)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.clf_report import *  # NOQA
>>> y_true = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3]
>>> y_pred = [1, 2, 1, 3, 1, 2, 2, 3, 2, 2, 3, 3, 2, 3, 3, 3, 1, 3]
>>> target_names = None
>>> sample_weight = None
>>> logs = []
>>> report = classification_report(y_true, y_pred, verbose=1, ascii_
↪only=True, log=logs.append)
>>> print('
```

‘.join(logs))

Ignore:

```
>>> size = 100
>>> rng = np.random.RandomState(0)
>>> p_classes = np.array([.90, .05, .05][0:2])
>>> p_classes = p_classes / p_classes.sum()
>>> p_wrong   = np.array([.03, .01, .02][0:2])
>>> y_true = testdata_ytrue(p_classes, p_wrong, size, rng)
```

(continues on next page)

(continued from previous page)

```
>>> rs = []
>>> for x in range(17):
>>>     p_wrong += .05
>>>     y_pred = testdata_ypred(y_true, p_wrong, rng)
>>>     report = classification_report(y_true, y_pred, verbose='hack')
>>>     rs.append(report)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> import pandas as pd
>>> df = pd.DataFrame(rs).drop(['raw'], axis=1)
>>> delta = df.subtract(df['target'], axis=0)
>>> sqrd_error = np.sqrt((delta ** 2).sum(axis=0))
>>> print('Error')
>>> print(sqrd_error.sort_values())
>>> ys = df.to_dict(orient='list')
>>> kwplot.multi_plot(ydata_list=ys)
```

kwcoco.metrics.clf_report.**ovr_classification_report**(*mc_y_true*, *mc_probs*, *target_names=None*, *sample_weight=None*, *metrics=None*, *verbose=0*, *remove_unsupported=False*, *log=None*)

One-vs-rest classification report

Parameters

- **mc_y_true** (*ndarray[int]*) – multiclass truth labels (integer label format). Shape [N].
- **mc_probs** (*ndarray*) – multiclass probabilities for each class. Shape [N x C].
- **target_names (Dict[int, str]** – mapping from int label to string name
- **sample_weight** (*ndarray*) – weight for each item. Shape [N].
- **metrics** (*List[str]*) – names of metrics to compute

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:sklearn)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.clf_report import *  # NOQA
>>> y_true = [1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0]
>>> y_probs = np.random.rand(len(y_true), max(y_true) + 1)
>>> target_names = None
>>> sample_weight = None
>>> verbose = True
>>> report = ovr_classification_report(y_true, y_probs)
>>> print(report['ave'])
auc      0.6541
ap       0.6824
kappa    0.0963
mcc      0.1002
brier    0.2214
```

(continues on next page)

(continued from previous page)

```
dtype: float64
>>> print(report['ovr'])
     auc      ap  kappa    mcc  brier  support  weight
0 0.6062 0.6161 0.0526 0.0598 0.2608        8  0.4444
1 0.5846 0.6014 0.0000 0.0000 0.2195        5  0.2778
2 0.8000 0.8693 0.2623 0.2652 0.1602        5  0.2778
```

Ignore:

```
>>> y_true = [1, 1, 1]
>>> y_probs = np.random.rand(len(y_true), 3)
>>> target_names = None
>>> sample_weight = None
>>> verbose = True
>>> report = ovr_classification_report(y_true, y_probs)
>>> print(report['ovr'])
```

kwcoco.metrics.confusion_vectors

Module Contents

Classes

ConfusionVectors	Stores information used to construct a confusion matrix. This includes
OneVsRestConfusionVectors	Container for multiple one-vs-rest binary confusion vectors
BinaryConfusionVectors	Stores information about a binary classification problem.
Measures	

Example

PerClass_Measures	

Functions

_stabalize_data(y_true, y_score, sample_weight, npad=7)	Adds ideally calibrated dummy values to curves with few positive examples.
populate_info(info)	

class kwcoco.metrics.confusion_vectors.**ConfusionVectors**(*cfsn_vecs*, *data*, *classes*, *probs=None*)

Bases: ubelt.NiceRepr

Stores information used to construct a confusion matrix. This includes corresponding vectors of predicted labels,

true labels, sample weights, etc…

Variables

- **data** (*DataFrameArray*) – should at least have keys true, pred, weight
- **classes** (*Sequence | CategoryTree*) – list of category names or category graph
- **probs** (*ndarray, optional*) – probabilities for each class

Example

```
>>> # xdoctest: IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data._pandas())
     pred  true   score  weight     iou  txs  pxs  gid
0       2     2 10.0000  1.0000  1.0000    0    4    0
1       2     2  7.5025  1.0000  1.0000    1    3    0
2       1     1  5.0050  1.0000  1.0000    2    2    0
3       3    -1  2.5075  1.0000 -1.0000   -1    1    0
4       2    -1  0.0100  1.0000 -1.0000   -1    0    0
5      -1     2  0.0000  1.0000 -1.0000    3   -1    0
6      -1     2  0.0000  1.0000 -1.0000    4   -1    0
7       2     2 10.0000  1.0000  1.0000    0    5    1
8       2     2  8.0020  1.0000  1.0000    1    4    1
9       1     1  6.0040  1.0000  1.0000    2    3    1
..    ...   ...     ...     ...     ...  ...  ...  ...
62     -1     2  0.0000  1.0000 -1.0000    7   -1    7
63     -1     3  0.0000  1.0000 -1.0000    8   -1    7
64     -1     1  0.0000  1.0000 -1.0000    9   -1    7
65      1    -1 10.0000  1.0000 -1.0000   -1    0    8
66      1     1  0.0100  1.0000  1.0000    0    1    8
67      3    -1 10.0000  1.0000 -1.0000   -1    3    9
68      2     2  6.6700  1.0000  1.0000    0    2    9
69      2     2  3.3400  1.0000  1.0000    1    1    9
70      3    -1  0.0100  1.0000 -1.0000   -1    0    9
71     -1     2  0.0000  1.0000 -1.0000    2   -1    9
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors
>>> cfsn_vecs = ConfusionVectors.demo(
>>>     nimgs=128, nboxes=(0, 10), n_fp=(0, 3), n_fn=(0, 3), classes=3)
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> measures = cx_to_binvecs.measures()['perclass']
>>> print('measures = {!r}'.format(measures))
measures = <PerClass_Measures({
    'cat_1': <Measures({'ap': 0.227, 'auc': 0.507, 'catname': cat_1, 'max_f1': f1=0.
↪45@0.47, 'nsupport': 788.000})>,
```

(continues on next page)

(continued from previous page)

```
    'cat_2': <Measures({'ap': 0.288, 'auc': 0.572, 'catname': cat_2, 'max_f1': f1=0.
↪51@0.43, 'nsupport': 788.000})>,
    'cat_3': <Measures({'ap': 0.225, 'auc': 0.484, 'catname': cat_3, 'max_f1': f1=0.
↪46@0.40, 'nsupport': 788.000})>,
}) at 0x7facf77bdfd0>
>>> kwplot.figure(fnum=1, doclf=True)
>>> measures.draw(key='pr', fnum=1, pnum=(1, 3, 1))
>>> measures.draw(key='roc', fnum=1, pnum=(1, 3, 2))
>>> measures.draw(key='mcc', fnum=1, pnum=(1, 3, 3))
...
```

__nice__(*cfsn_vecs*)

__json__(*self*)

Serialize to json

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import ConfusionVectors
>>> self = ConfusionVectors.demo(n_imgs=1, classes=2, n_fp=0, nboxes=1)
>>> state = self.__json__()
>>> print('state = {}'.format(ub.repr2(state, nl=2, precision=2, align=1)))
>>> recon = ConfusionVectors.from_json(state)
```

classmethod from_json(*cls*, *state*)

classmethod demo(*cfsn_vecs*, ***kw*)

Parameters ****kwargs** – See *kwcoco.metrics.DetectionMetrics.demo()*

Returns ConfusionVectors

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> print('cx_to_binvecs = {!r}'.format(cx_to_binvecs))
```

classmethod from_arrays(*ConfusionVectors*, *true*, *pred=None*, *score=None*, *weight=None*, *probs=None*, *classes=None*)

Construct confusion vector data structure from component arrays

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import kwarray
>>> classes = ['person', 'vehicle', 'object']
>>> rng = kwarray.ensure_rng(0)
>>> true = (rng.rand(10) * len(classes)).astype(int)
>>> probs = rng.rand(len(true), len(classes))
>>> cfsn_vecs = ConfusionVectors.from_arrays(true=true, probs=probs,␣
↪classes=classes)
>>> cfsn_vecs.confusion_matrix()
pred     person  vehicle  object
real
person        0        0       0
vehicle       2        4       1
object        2        1       0
```

confusion_matrix(*cfsn_vecs*, *compress=False*)

Builds a confusion matrix from the confusion vectors.

Parameters **compress** (*bool, default=False*) – if True removes rows / columns with no entries

Returns

cm [the labeled confusion matrix]

(Note: we should write a efficient replacement for this use case. #remove_pandas)

Return type pd.DataFrame

CommandLine: xdoctest -m kwcoco.metrics.confusion_vectors ConfusionVectors.confusion_matrix

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), n_fn=(0, 1), classes=3, cls_
↪noise=.2)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> cm = cfsn_vecs.confusion_matrix()
...
>>> print(cm.to_string(float_format=lambda x: '%.2f' % x))
pred        background  cat_1  cat_2  cat_3
real
background        0.00   1.00   2.00   3.00
cat_1             3.00  12.00   0.00   0.00
cat_2             3.00   0.00  14.00   0.00
cat_3             2.00   0.00   0.00  17.00
```

coarsen(*cfsn_vecs*, *cxs*)

Creates a coarsened set of vectors

Returns ConfusionVectors

binarize_classless(*cfsn_vecs*, *negative_classes=None*)

Creates a binary representation useful for measuring the performance of detectors. It is assumed that scores of "positive" classes should be high and "negative" clases should be low.

Parameters **negative_classes** (*List[str | int]*) – list of negative class names or idxs, by default chooses any class with a true class index of -1. These classes should ideally have low scores.

Returns BinaryConfusionVectors

Notes

The "classlessness" of this depends on the compat="all" argument being used when constructing confusion vectors, otherwise it becomes something like a macro-average because the class information was used in deciding which true and predicted boxes were allowed to match.

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), n_fn=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> class_idxs = list(dmet.classes.node_to_idx.values())
>>> binvecs = cfsn_vecs.binarize_classless()
```

binarize_ovr(*cfsn_vecs*, *mode=1*, *keyby='name'*, *ignore_classes={'ignore'}*, *approx=0*)

Transforms cfsn_vecs into one-vs-rest BinaryConfusionVectors for each category.

Parameters

- **mode** (*int, default=1*) – 0 for heirarchy aware or 1 for voc like. MODE 0 IS PROBABLY BROKEN
- **keyby** (*int | str*) – can be cx or name
- **ignore_classes** (*Set[str]*) – category names to ignore
- **approx** (*bool, default=0*) – if True try and approximate missing scores otherwise assume they are irrecoverable and use -inf

Returns

which behaves like Dict[int, BinaryConfusionVectors]: cx_to_binvecs

Return type *OneVsRestConfusionVectors*

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> catname_to_binvecs = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('catname_to_binvecs = {!r}'.format(catname_to_binvecs))
```

cfsn_vecs.data.pandas() catname_to_binvecs.cx_to_binvecs['class_1'].data.pandas()

Notes

classification_report(*cfsn_vecs*, *verbose=0*)
Build a classification report with various metrics.

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> report = cfsn_vecs.classification_report(verbose=1)
```

class kwcoco.metrics.confusion_vectors.**OneVsRestConfusionVectors**(*cx_to_binvecs*, *classes*)
Bases: ubelt.NiceRepr

Container for multiple one-vs-rest binary confusion vectors

Variables

- **cx_to_binvecs** –
- **classes** –

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> self = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('self = {!r}'.format(self))
```

__nice__(*self*)

classmethod demo(*cls*)

Parameters ****kwargs** – See *kwcoco.metrics.DetectionMetrics.demo()*

Returns ConfusionVectors

keys(*self*)

__getitem__(*self*, *cx*)

measures(*self*, *stabalize_thresh=7*, *fp_cutoff=None*, *monotonic_ppv=True*, *ap_method='pycocotools'*)
Creates binary confusion measures for every one-versus-rest category.

Parameters

- **stabalize_thresh** (*int, default=7*) – if fewer than this many data points inserts dummy stabalization data so curves can still be drawn.
- **fp_cutoff** (*int, default=None*) – maximum number of false positives in the truncated roc curves. None is equivalent to `float('inf')`
- **monotonic_ppv** (*bool, default=True*) – if True ensures that precision is always increasing as recall decreases. This is done in pycocotools scoring, but I'm not sure its a good idea.

SeeAlso: *BinaryConfusionVectors.measures()*

Example

```
>>> self = OneVsRestConfusionVectors.demo()
>>> thresh_result = self.measures()['perclass']
```

abstract ovr_classification_report(*self*)

class kwcoco.metrics.confusion_vectors.**BinaryConfusionVectors**(*data*, *cx=None*, *classes=None*)

Bases: ubelt.NiceRepr

Stores information about a binary classification problem. This is always with respect to a specific class, which is given by *cx* and *classes*.

The *data* DataFrameArray must contain *is_true* - if the row is an instance of class *classes[cx]* *pred_score* - the predicted probability of class *classes[cx]*, and *weight* - sample weight of the example

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=10)
>>> print('self = {!r}'.format(self))
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=0)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=1)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=2)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

classmethod demo(*cls*, *n=10*, *p_true=0.5*, *p_error=0.2*, *p_miss=0.0*, *rng=None*)

Create random data for tests

Parameters

- **n** (*int*) – number of rows
- **p_true** (*int*) – fraction of real positive cases
- **p_error** (*int*) – probability of making a recoverable mistake
- **p_miss** (*int*) – probability of making a unrecoverable mistake
- **rng** (*int | RandomState*) – random seed / state

Returns BinaryConfusionVectors

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn = BinaryConfusionVectors.demo(n=1000, p_error=0.1, p_miss=0.1)
>>> measures = cfsn.measures()
>>> print('measures = {}'.format(ub.repr2(measures, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, pnum=(1, 2, 1))
>>> measures.draw('pr')
>>> kwplot.figure(fnum=1, pnum=(1, 2, 2))
>>> measures.draw('roc')
```

property catname(*self*)

__nice__(*self*)

__len__(*self*)

measures(*self*, *stabalize_thresh=7*, *fp_cutoff=None*, *monotonic_ppv=True*, *ap_method='pycocotools'*)

Get statistics (F1, G1, MCC) versus thresholds

Parameters

- **stabalize_thresh** (*int, default=7*) – if fewer than this many data points inserts dummy stabalization data so curves can still be drawn.
- **fp_cutoff** (*int, default=None*) – maximum number of false positives in the truncated roc curves. `None` is equivalent to `float('inf')`
- **monotonic_ppv** (*bool, default=True*) – if True ensures that precision is always increasing as recall decreases. This is done in pycocotools scoring, but I'm not sure its a good idea.

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=0)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=1, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=3, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=100, p_true=0.5, p_error=0.5, p_miss=0.
↪3)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> print('measures = {}'.format(ub.repr2(ub.odict(self.measures()))))
```

Ignore:

import matplotlib.cm as cm # kwargs = {} # cmap = kwargs.get('cmap', mpl.cm.coolwarm) # n = len(x) # xgrid = np.tile(x[None, :], (n, 1)) # ygrid = np.tile(y[None, :], (n, 1)) # zdata = np.tile(z[None, :], (n, 1)) # ax.contour(xgrid, ygrid, zdata, zdir='x', cmap=cmap) # ax.contour(xgrid, ygrid, zdata, zdir='y', cmap=cmap) # ax.contour(xgrid, ygrid, zdata, zdir='z', cmap=cmap)

References

https://en.wikipedia.org/wiki/Confusion_matrix https://en.wikipedia.org/wiki/Precision_and_recall https://en.wikipedia.org/wiki/Matthews_correlation_coefficient

Ignore: self.measures().summary_plot() import xdev globals().update(xdev.get_func_kwargs(BinaryConfusionVectors.mea

_binary_clf_curves(*self*, *stabalize_thresh=7*, *fp_cutoff=None*)

Compute TP, FP, TN, and FN counts for this binary confusion vector.

Code common to ROC, PR, and threshold measures, computes the elements of the binary confusion matrix at all relevant operating point thresholds.

Parameters

- **stabalize_thresh** (*int*) – if fewer than this many data points insert stabalization data.
- **fp_cutoff** (*int*) – maximum number of false positives

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=1, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

```
>>> self = BinaryConfusionVectors.demo(n=0, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

```
>>> self = BinaryConfusionVectors.demo(n=100, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

Ignore: import xdev globals().update(xdev.get_func_kwargs(BinaryConfusionVectors._binary_clf_curves)) >>> self = BinaryConfusionVectors.demo(n=10, p_true=0.7, p_error=0.3, p_miss=0.2) >>> print('measures = {}'.format(ub.repr2(self._binary_clf_curves()))) >>> info = self.measures() >>> info = ub.dict_isect(info, ['tpr', 'fpr', 'ppv', 'fp_count']) >>> print('measures = {}'.format(ub.repr2(ub.odict(i))))

draw_distribution(*self*)

_3dplot(*self*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 2), nimgs=256, nboxes=(0, 10),
>>>     bbox_noise=10,
>>>     classes=1)
>>> cfsn_vecs = dmet.confusion_vectors()
```

(continues on next page)

(continued from previous page)

```
>>> self = bin_cfsn = cfsn_vecs.binarize_classless()
>>> #dmet.summarize(plot=True)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=3)
>>> self._3dplot()
```

class kwcoco.metrics.confusion_vectors.**Measures**(*info*)

Bases: ubelt.NiceRepr, *kwcoco.metrics.util.DictProxy*

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> binvecs = BinaryConfusionVectors.demo(n=100, p_error=0.5)
>>> self = binvecs.measures()
>>> print('self = {!r}'.format(self))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.draw(doclf=True)
>>> self.draw(key='pr',  pnum=(1, 2, 1))
>>> self.draw(key='roc', pnum=(1, 2, 2))
>>> kwplot.show_if_requested()
```

property catname(*self*)

__nice__(*self*)

reconstruct(*self*)

classmethod from_json(*cls*, *state*)

__json__(*self*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> binvecs = BinaryConfusionVectors.demo(n=10, p_error=0.5)
>>> self = binvecs.measures()
>>> info = self.__json__()
>>> print('info = {}'.format(ub.repr2(info, nl=1)))
>>> populate_info(info)
>>> print('info = {}'.format(ub.repr2(info, nl=1)))
>>> recon = Measures.from_json(info)
```

summary(*self*)

draw(*self*, *key=None*, *prefix=''*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

summary_plot(*self*, *fnum=1*, *title=''*, *subplots='auto'*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo(n=3, p_error=0.5)
>>> binvecs = cfsn_vecs.binarize_classless()
>>> self = binvecs.measures()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.summary_plot()
>>> kwplot.show_if_requested()
```

class kwcoco.metrics.confusion_vectors.**PerClass_Measures**(*cx_to_info*)

Bases: ubelt.NiceRepr, *kwcoco.metrics.util.DictProxy*

__nice__(*self*)

summary(*self*)

classmethod from_json(*cls*, *state*)

__json__(*self*)

draw(*self*, *key='mcc'*, *prefix=''*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

draw_roc(*self*, *prefix=''*, ***kw*)

draw_pr(*self*, *prefix=''*, ***kw*)

summary_plot(*self*, *fnum=1*, *title=''*, *subplots='auto'*)

CommandLine: python ~/code/kwcoco/kwcoco/metrics/confusion_vectors.py PerClass_Measures.summary_plot –show

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 3), nimgs=32, nboxes=(0, 32),
>>>     classes=3, rng=0, newstyle=1, box_noise=0.7, cls_noise=0.2, score_
↪noise=0.3, with_probs=False)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name', ignore_classes=['vector',
↪'raster'])
>>> self = ovr_cfsn.measures()['perclass']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> import seaborn as sns
>>> sns.set()
>>> self.summary_plot(title='demo summary_plot ovr', subplots=['pr', 'roc'])
>>> kwplot.show_if_requested()
>>> self.summary_plot(title='demo summary_plot ovr', subplots=['mcc', 'acc'],␣
↪fnum=2)
```

kwcoco.metrics.confusion_vectors.**_stabalize_data**(*y_true*, *y_score*, *sample_weight*, *npad=7*)

Adds ideally calibrated dummy values to curves with few positive examples. This acts somewhat like a Baysian prior and smooths out the curve.

Example

y_score = np.array([0.5, 0.6]) y_true = np.array([1, 1]) sample_weight = np.array([1, 1]) npad = 7 _stabalize_data(y_true, y_score, sample_weight, npad=npad)

kwcoco.metrics.confusion_vectors.**populate_info**(*info*)

kwcoco.metrics.detect_metrics

Module Contents

Classes

DetectionMetrics	Object that computes associations between detections and can convert them

Functions

_demo_construct_probs(pred_cxs, pred_scores, classes, rng, hacked=1)	Constructs random probabilities for demo data
pycocotools_confusion_vectors(dmet, evaler, iou_thresh=0.5, verbose=0)	
	Example
eval_detections_cli(**kw)	DEPRECATED USE *kwcoco eval* instead
_summarize(self, ap=1, iouThr=None, areaRngLbl='all', maxDets=100)	
pct_summarize2(self)	

class kwcoco.metrics.detect_metrics.**DetectionMetrics**(*dmet*, *classes=None*)

Bases: ubelt.NiceRepr

Object that computes associations between detections and can convert them into sklearn-compatible representations for scoring.

Variables

- **gid_to_true_dets** (*Dict*) – maps image ids to truth
- **gid_to_pred_dets** (*Dict*) – maps image ids to predictions
- **classes** (CategoryTree) – category coder

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), classes=8, score_noise=0.9,␣
↪hacked=False)
>>> print(dmet.score_kwcoco(bias=0, compat='mutex', prioritize='iou')['mAP'])
...
>>> # NOTE: IN GENERAL NETHARN AND VOC ARE NOT THE SAME
>>> print(dmet.score_voc(bias=0)['mAP'])
0.8582...
>>> #print(dmet.score_coco()['mAP'])
```

score_coco

clear(*dmet*)

__nice__(*dmet*)

classmethod from_coco(*DetectionMetrics*, *true_coco*, *pred_coco*, *gids=None*, *verbose=0*)

Create detection metrics from two coco files representing the truth and predictions.

Parameters

- **true_coco** (*kwcoco.CocoDataset*)
- **pred_coco** (*kwcoco.CocoDataset*)

Example

```
>>> import kwcoco
>>> from kwcoco.demo.perterb import perterb_coco
>>> true_coco = kwcoco.CocoDataset.demo('shapes')
>>> perterbkw = dict(box_noise=0.5, cls_noise=0.5, score_noise=0.5)
>>> pred_coco = perterb_coco(true_coco, **perterbkw)
>>> self = DetectionMetrics.from_coco(true_coco, pred_coco)
>>> self.score_voc()
```

_register_imagename(*dmet*, *imgname*, *gid=None*)

add_predictions(*dmet*, *pred_dets*, *imgname=None*, *gid=None*)
Register/Add predicted detections for an image

Parameters

- **pred_dets** (*Detections*) – predicted detections
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int, optional*) – the integer image id if known

add_truth(*dmet*, *true_dets*, *imgname=None*, *gid=None*)
Register/Add groundtruth detections for an image

Parameters

- **true_dets** (*Detections*) – groundtruth
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int, optional*) – the integer image id if known

true_detections(*dmet*, *gid*)
gets Detections representation for groundtruth in an image

pred_detections(*dmet*, *gid*)
gets Detections representation for predictions in an image

confusion_vectors(*dmet*, *iou_thresh=0.5*, *bias=0*, *gids=None*, *compat='mutex'*, *prioritize='iou'*, *ignore_classes='ignore'*, *background_class=ub.NoParam*, *verbose='auto'*, *workers=0*, *track_probs='try'*, *max_dets=None*)
Assigns predicted boxes to the true boxes so we can transform the detection problem into a classification problem for scoring.

Parameters

- **iou_thresh** (*float | List[float], default=0.5*) – bounding box overlap iou threshold required for assignment if a list, then return type is a dict
- **bias** (*float, default=0.0*) – for computing bounding box overlap, either 1 or 0
- **gids** (*List[int], default=None*) – which subset of images ids to compute confusion metrics on. If not specified all images are used.
- **compat** (*str, default='all'*) – can be ('ancestors' | 'mutex' | 'all'). determines which pred boxes are allowed to match which true boxes. If 'mutex', then pred boxes can only match true boxes of the same class. If 'ancestors', then pred boxes can match true boxes that match or have a coarser label. If 'all', then any pred can match any true, regardless of its category label.

- **prioritize** (*str, default='iou'*) – can be ('iou' | 'class' | 'correct') determines which box to assign to if mutiple true boxes overlap a predicted box. if prioritize is iou, then the true box with maximum iou (above iou_thresh) will be chosen. If prioritize is class, then it will prefer matching a compatible class above a higher iou. If prioritize is correct, then ancestors of the true class are preferred over descendents of the true class, over unreleated classes.
- **ignore_classes** (*set, default={'ignore'}*) – class names indicating ignore regions
- **background_class** (*str, default=ub.NoParam*) – Name of the background class. If unspecified we try to determine it with heuristics. A value of None means there is no background class.
- **verbose** (*int, default='auto'*) – verbosity flag. In auto mode, verbose=1 if len(gids) > 1000.
- **workers** (*int, default=0*) – number of parallel assignment processes
- **track_probs** (*str, default='try'*) – can be 'try', 'force', or False. if truthy, we assume probabilities for multiple classes are available.

Returns ConfusionVectors | Dict[float, ConfusionVectors]

Ignore: globals().update(xdev.get_func_kwargs(dmet.confusion_vectors))

Example

```
>>> dmet = DetectionMetrics.demo(nimgs=30, classes=3,
>>>                              nboxes=10, n_fp=3, box_noise=10,
>>>                              with_probs=False)
>>> iou_to_cfsn = dmet.confusion_vectors(iou_thresh=[0.3, 0.5, 0.9])
>>> for t, cfsn in iou_to_cfsn.items():
>>>     print('t = {!r}'.format(t))
...     print(cfsn.binarize_ovr().measures())
...     print(cfsn.binarize_classless().measures())
```

score_kwant(*dmet*, *iou_thresh=0.5*)
Scores the detections using kwant

score_kwcoco(*dmet*, *iou_thresh=0.5*, *bias=0*, *gids=None*, *compat='all'*, *prioritize='iou'*)
our scoring method

score_voc(*dmet*, *iou_thresh=0.5*, *bias=1*, *method='voc2012'*, *gids=None*, *ignore_classes='ignore'*)
score using voc method

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), classes=8,
>>>     score_noise=.5)
>>> print(dmet.score_voc()['mAP'])
0.9399...
```

_to_coco(*dmet*)
Convert to a coco representation of truth and predictions

with inverse aid mappings

score_pycocotools(*dmet*, *with_evaler=False*, *with_confusion=False*, *verbose=0*, *iou_thresholds=None*)
score using ms-coco method

Returns dictionary with pct info

Return type Dict

Example

```
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> from kwcoco.metrics.detect_metrics import *
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 3), n_fn=(0, 1), n_fp=(0, 1), classes=8, with_
↪probs=False)
>>> pct_info = dmet.score_pycocotools(verbose=1,
>>>                                   with_evaler=True,
>>>                                   with_confusion=True,
>>>                                   iou_thresholds=[0.5, 0.9])
>>> evaler = pct_info['evaler']
>>> iou_to_cfsn_vecs = pct_info['iou_to_cfsn_vecs']
>>> for iou_thresh in iou_to_cfsn_vecs.keys():
>>>     print('iou_thresh = {!r}'.format(iou_thresh))
>>>     cfsn_vecs = iou_to_cfsn_vecs[iou_thresh]
>>>     ovr_measures = cfsn_vecs.binarize_ovr().measures()
>>>     print('ovr_measures = {}'.format(ub.repr2(ovr_measures, nl=1,␣
↪precision=4)))
```

Notes

by default pycocotools computes average precision as the literal average of computed precisions at 101 uniformly spaced recall thresholds.

pycocoutils seems to only allow predictions with the same category as the truth to match those truth objects. This should be the same as calling dmet.confusion_vectors with compat = mutex

pycocoutils does not take into account the fact that each box often has a score for each category.

pycocoutils will be incorrect if any annotation has an id of 0

a major difference in the way kwcoco scores versus pycocoutils is the calculation of AP. The assignment between truth and predicted detections produces similar enough results. Given our confusion vectors we use the scikit-learn definition of AP, whereas pycocoutils seems to compute precision and recall — more or less correctly — but then it resamples the precision at various specified recall thresholds (in the *accumulate* function, specifically how *pr* is resampled into the *q* array). This can lead to a large difference in reported scores.

pycocoutils also smooths out the precision such that it is monotonic decreasing, which might not be the best idea.

pycocotools area ranges are inclusive on both ends, that means the "small" and "medium" truth selections do overlap somewhat.

classmethod demo(*cls*, ***kwargs*)
Creates random true boxes and predicted boxes that have some noisy offset from the truth.

Kwargs:

classes (int, default=1): class list or the number of foreground classes.

nimgs (int, default=1): number of images in the coco datasts. nboxes (int, default=1): boxes per image. n_fp (int, default=0): number of false positives. n_fn (int, default=0): number of false negatives. box_noise (float, default=0): std of a normal distribution used to

perterb both box location and box size.

cls_noise (float, default=0): probability that a class label will change. Must be within 0 and 1.

anchors (ndarray, default=None): used to create random boxes null_pred (bool, default=0):

if True, predicted classes are returned as null, which means only localization scoring is suitable.

with_probs (bool, default=1): if True, includes per-class probabilities with predictions

CommandLine: xdoctest -m kwcoco.metrics.detect_metrics DetectionMetrics.demo:2 –show

Example

```
>>> kwargs = {}
>>> # Seed the RNG
>>> kwargs['rng'] = 0
>>> # Size parameters determine how big the data is
>>> kwargs['nimgs'] = 5
>>> kwargs['nboxes'] = 7
>>> kwargs['classes'] = 11
>>> # Noise parameters perterb predictions further from the truth
>>> kwargs['n_fp'] = 3
>>> kwargs['box_noise'] = 0.1
>>> kwargs['cls_noise'] = 0.5
>>> dmet = DetectionMetrics.demo(**kwargs)
>>> print('dmet.classes = {}'.format(dmet.classes))
dmet.classes = <CategoryTree(nNodes=12, maxDepth=3, maxBreadth=4...)>
>>> # Can grab kwimage.Detection object for any image
>>> print(dmet.true_detections(gid=0))
<Detections(4)>
>>> print(dmet.pred_detections(gid=0))
<Detections(7)>
```

Example

```
>>> # Test case with null predicted categories
>>> dmet = DetectionMetrics.demo(nimgs=30, null_pred=1, classes=3,
>>>                              nboxes=10, n_fp=3, box_noise=0.1,
>>>                              with_probs=False)
>>> dmet.gid_to_pred_dets[0].data
>>> dmet.gid_to_true_dets[0].data
>>> cfsn_vecs = dmet.confusion_vectors()
>>> binvecs_ovr = cfsn_vecs.binarize_ovr()
>>> binvecs_per = cfsn_vecs.binarize_classless()
```

(continues on next page)

(continued from previous page)

```
>>> measures_per = binvecs_per.measures()
>>> measures_ovr = binvecs_ovr.measures()
>>> print('measures_per = {!r}'.format(measures_per))
>>> print('measures_ovr = {!r}'.format(measures_ovr))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> measures_ovr['perclass'].draw(key='pr', fnum=2)
```

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 1), nimgs=32, nboxes=(0, 16),
>>>     classes=3, rng=0, newstyle=1, box_noise=0.5, cls_noise=0.0, score_
↪noise=0.3, with_probs=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> summary = dmet.summarize(plot=True, title='DetectionMetrics summary demo',␣
↪with_ovr=True, with_bin=False)
>>> summary['bin_measures']
>>> kwplot.show_if_requested()
```

summarize(*dmet*, *out_dpath=None*, *plot=False*, *title=''*, *with_bin='auto'*, *with_ovr='auto'*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 128), n_fn=(0, 4), nimgs=512, nboxes=(0, 32),
>>>     classes=3, rng=0)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dmet.summarize(plot=True, title='DetectionMetrics summary demo')
>>> kwplot.show_if_requested()
```

kwcoco.metrics.detect_metrics.**_demo_construct_probs**(*pred_cxs*, *pred_scores*, *classes*, *rng*, *hacked=1*)

Constructs random probabilities for demo data

kwcoco.metrics.detect_metrics.**pycocotools_confusion_vectors**(*dmet*, *evaler*, *iou_thresh=0.5*, *verbose=0*)

Example

```
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> from kwcoco.metrics.detect_metrics import *
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 3), n_fn=(0, 1), n_fp=(0, 1), classes=8, with_
↪probs=False)
>>> coco_scores = dmet.score_pycocotools(with_evaler=True)
>>> evaler = coco_scores['evaler']
>>> cfsn_vecs = pycocotools_confusion_vectors(dmet, evaler, verbose=1)
```

kwcoco.metrics.detect_metrics.**eval_detections_cli**(***kw*)

DEPRECATED USE *kwcoco eval* instead

CommandLine: xdoctest -m ~/code/kwcoco/kwcoco/metrics/detect_metrics.py eval_detections_cli

kwcoco.metrics.detect_metrics.**_summarize**(*self*, *ap=1*, *iouThr=None*, *areaRngLbl='all'*, *maxDets=100*)

kwcoco.metrics.detect_metrics.**pct_summarize2**(*self*)

kwcoco.metrics.drawing

Module Contents

Functions

draw_perclass_roc(cx_to_info, classes=None, prefix='', fnum=1, fp_axis='count', **kw)	**Parameters** • **cx_to_info** (*PerClass_Measures* | *Dict*)
inty_display(val, eps=1e-08, ndigits=2)	Make a number as inty as possible
_realpos_label_suffix(info)	Creates a label suffix that indicates the number of real positive cases
draw_perclass_prcurve(cx_to_info, classes=None, prefix='', fnum=1, **kw)	**Parameters** **cx_to_info** (*PerClass_Measures* | *Dict*)
draw_perclass_thresholds(cx_to_info, key='mcc', classes=None, prefix='', fnum=1, **kw)	**Parameters** **cx_to_info** (*PerClass_Measures* | *Dict*)
draw_roc(info, prefix='', fnum=1, **kw)	**Parameters** **info** (*Measures* | *Dict*)
draw_prcurve(info, prefix='', fnum=1, **kw)	Draws a single pr curve.
draw_threshold_curves(info, keys=None, prefix='', fnum=1, **kw)	**Parameters** **info** (*Measures* | *Dict*)

kwcoco.metrics.drawing.**draw_perclass_roc**(*cx_to_info*, *classes=None*, *prefix=''*, *fnum=1*, *fp_axis='count'*, ***kw*)

Parameters

- **cx_to_info** (*PerClass_Measures | Dict*)
- **fp_axis** (*str*) – can be count or rate

kwcoco.metrics.drawing.**inty_display**(*val*, *eps=1e-08*, *ndigits=2*)
Make a number as inty as possible

kwcoco.metrics.drawing.**_realpos_label_suffix**(*info*)
Creates a label suffix that indicates the number of real positive cases versus the total amount of cases considered for an evaluation curve.

Parameters **info** (*Dict*) – with keys, nsuppert, realpos_total

Example

```
>>> info = {'nsupport': 10, 'realpos_total': 10}
>>> _realpos_label_suffix(info)
10/10
>>> info = {'nsupport': 10.0, 'realpos_total': 10.0}
>>> _realpos_label_suffix(info)
10/10
>>> info = {'nsupport': 10.3333, 'realpos_total': 10.22222}
>>> _realpos_label_suffix(info)
10.22/10.33
>>> info = {'nsupport': 10.000000001, 'realpos_total': None}
>>> _realpos_label_suffix(info)
10
>>> info = {'nsupport': 10.009}
>>> _realpos_label_suffix(info)
10.01
```

kwcoco.metrics.drawing.**draw_perclass_prcurve**(*cx_to_info*, *classes=None*, *prefix=''*, *fnum=1*, ***kw*)

Parameters **cx_to_info** (*PerClass_Measures | Dict*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=3, nboxes=(0, 10), n_fp=(0, 3), n_fn=(0, 2), classes=3, score_noise=0.
↪1, box_noise=0.1, with_probs=False)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data.pandas())
>>> classes = cfsn_vecs.classes
>>> cx_to_info = cfsn_vecs.binarize_ovr().measures()['perclass']
>>> print('cx_to_info = {}'.format(ub.repr2(cx_to_info, nl=1)))
```

(continues on next page)

(continued from previous page)

```
>>> import kwplot
>>> kwplot.autompl()
>>> draw_perclass_prcurve(cx_to_info, classes)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```

Ignore: from kwcoco.metrics.drawing import * # NOQA import xdev globals().update(xdev.get_func_kwargs(draw_perclass_prcurve))

kwcoco.metrics.drawing.**draw_perclass_thresholds**(*cx_to_info*, *key='mcc'*, *classes=None*, *prefix=''*, *fnum=1*, ***kw*)

Parameters **cx_to_info** (*PerClass_Measures | Dict*)

Notes

Each category is inspected independently of one another, there is no notion of confusion.

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> from kwcoco.metrics import ConfusionVectors
>>> cfsn_vecs = ConfusionVectors.demo()
>>> classes = cfsn_vecs.classes
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> cx_to_info = ovr_cfsn.measures()['perclass']
>>> import kwplot
>>> kwplot.autompl()
>>> key = 'mcc'
>>> draw_perclass_thresholds(cx_to_info, key, classes)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```

kwcoco.metrics.drawing.**draw_roc**(*info*, *prefix=''*, *fnum=1*, ***kw*)

Parameters **info** (*Measures | Dict*)

Note: There needs to be enough negative examples for using ROC to make any sense!

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot, module:seaborn)
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(nimgs=30, null_pred=1, classes=3,
>>>                              nboxes=10, n_fp=10, box_noise=0.3,
>>>                              with_probs=False)
>>> dmet.true_detections(0).data
>>> cfsn_vecs = dmet.confusion_vectors(compat='mutex', prioritize='iou', bias=0)
>>> print(cfsn_vecs.data._pandas().sort_values('score'))
>>> classes = cfsn_vecs.classes
>>> info = ub.peek(cfsn_vecs.binarize_ovr().measures()['perclass'].values())
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> draw_roc(info)
>>> kwplot.show_if_requested()
```

kwcoco.metrics.drawing.**draw_prcurve**(*info*, *prefix=''*, *fnum=1*, ***kw*)

Draws a single pr curve.

Parameters **info** (*Measures | Dict*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
```

```
>>> classes = cfsn_vecs.classes
>>> info = cfsn_vecs.binarize_classless().measures()
>>> import kwplot
>>> kwplot.autompl()
>>> draw_prcurve(info)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```

kwcoco.metrics.drawing.**draw_threshold_curves**(*info*, *keys=None*, *prefix=''*, *fnum=1*, ***kw*)

Parameters **info** (*Measures | Dict*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import sys, ubelt
>>> sys.path.append(ubelt.expandpath('~/code/kwcoco'))
>>> from kwcoco.metrics.drawing import *  # NOQA
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> info = cfsn_vecs.binarize_classless().measures()
>>> keys = None
>>> import kwplot
>>> kwplot.autompl()
>>> draw_threshold_curves(info, keys)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.show_if_requested()
```

kwcoco.metrics.functional

Module Contents

Functions

fast_confusion_matrix(y_true, y_pred, n_labels, sample_weight=None)	faster version of sklearn confusion matrix that avoids the
_truncated_roc(y_df, bg_idx=-1, fp_cutoff=None)	Computes truncated ROC info
_pr_curves(y)	Compute a PR curve from a method
_average_precision(tpr, ppv)	Compute average precision of a binary PR curve. This is simply the area

kwcoco.metrics.functional.**fast_confusion_matrix**(*y_true*, *y_pred*, *n_labels*, *sample_weight=None*)

faster version of sklearn confusion matrix that avoids the expensive checks and label rectification

Parameters

- **y_true** (*ndarray[int]*) – ground truth class label for each sample
- **y_pred** (*ndarray[int]*) – predicted class label for each sample
- **n_labels** (*int*) – number of labels
- **sample_weight** (*ndarray[int|float]*) – weight of each sample

Returns matrix where rows represent real and cols represent pred and the value at each cell is the total amount of weight

Return type ndarray[int64|float64, dim=2]

Example

```
>>> y_true = np.array([0, 0, 0, 0, 1, 1, 1, 0,  0, 1])
>>> y_pred = np.array([0, 0, 0, 0, 0, 0, 0, 1,  1, 1])
>>> fast_confusion_matrix(y_true, y_pred, 2)
array([[4, 2],
       [3, 1]])
>>> fast_confusion_matrix(y_true, y_pred, 2).ravel()
array([4, 2, 3, 1])
```

kwcoco.metrics.functional.**_truncated_roc**(*y_df*, *bg_idx=- 1*, *fp_cutoff=None*)

Computes truncated ROC info

kwcoco.metrics.functional.**_pr_curves**(*y*)

Compute a PR curve from a method

Parameters **y** (*pd.DataFrame | DataFrameArray*) – output of detection_confusions

Returns Tuple[float, ndarray, ndarray]

Example

```
>>> # xdoctest: +REQUIRES(module:sklearn)
>>> import pandas as pd
>>> y1 = pd.DataFrame.from_records([
>>>     {'pred': 0, 'score': 10.00, 'true': -1, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.65, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  8.64, 'true': -1, 'weight': 1.00},
>>>     {'pred': 0, 'score':  3.97, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.68, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  5.06, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  0.25, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.75, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  8.52, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  5.20, 'true':  0, 'weight': 1.00},
>>> ])
>>> import kwcoco as nh
>>> import kwarray
>>> y2 = kwarray.DataFrameArray(y1)
>>> _pr_curves(y2)
>>> _pr_curves(y1)
```

kwcoco.metrics.functional.**_average_precision**(*tpr*, *ppv*)

Compute average precision of a binary PR curve. This is simply the area under the curve.

Parameters

- **tpr** (*ndarray*) – true positive rate - aka recall
- **ppv** (*ndarray*) – positive predictive value - aka precision

kwcoco.metrics.sklearn_alts

Faster pure-python versions of sklearn functions that avoid expensive checks and label rectifications. It is assumed that all labels are consecutive non-negative integers.

Module Contents

Functions

confusion_matrix(y_true, y_pred, n_labels=None, labels=None, sample_weight=None)	faster version of sklearn confusion matrix that avoids the
global_accuracy_from_confusion(cfsn)	
class_accuracy_from_confusion(cfsn)	
_binary_clf_curve2(y_true, y_score, pos_label=None, sample_weight=None)	MODIFIED VERSION OF SCIKIT-LEARN API

kwcoco.metrics.sklearn_alts.**confusion_matrix**(*y_true*, *y_pred*, *n_labels=None*, *labels=None*, *sample_weight=None*)

faster version of sklearn confusion matrix that avoids the expensive checks and label rectification

Runs in about 0.7ms

Returns matrix where rows represent real and cols represent pred

Return type ndarray

Example

```
>>> y_true = np.array([0, 0, 0, 0, 1, 1, 1, 0,  0, 1])
>>> y_pred = np.array([0, 0, 0, 0, 0, 0, 0, 1,  1, 1])
>>> confusion_matrix(y_true, y_pred, 2)
array([[4, 2],
       [3, 1]])
>>> confusion_matrix(y_true, y_pred, 2).ravel()
array([4, 2, 3, 1])
```

Benchmarks: import ubelt as ub y_true = np.random.randint(0, 2, 10000) y_pred = np.random.randint(0, 2, 10000)

n = 1000 for timer in ub.Timerit(n, bestof=10, label='py-time'):

sample_weight = [1] * len(y_true) confusion_matrix(y_true, y_pred, 2, sample_weight=sample_weight)

for timer in ub.Timerit(n, bestof=10, label='np-time'): sample_weight = np.ones(len(y_true), dtype=int) confusion_matrix(y_true, y_pred, 2, sample_weight=sample_weight)

kwcoco.metrics.sklearn_alts.**global_accuracy_from_confusion**(*cfsn*)

kwcoco.metrics.sklearn_alts.**class_accuracy_from_confusion**(*cfsn*)

kwcoco.metrics.sklearn_alts.**_binary_clf_curve2**(*y_true*, *y_score*, *pos_label=None*, *sample_weight=None*)

MODIFIED VERSION OF SCIKIT-LEARN API

Calculate true and false positives per binary classification threshold.

Parameters

- **y_true** (*array, shape = [n_samples]*) – True targets of binary classification
- **y_score** (*array, shape = [n_samples]*) – Estimated probabilities or decision function
- **pos_label** (*int or str, default=None*) – The label of the positive class
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns

- **fps** (*array, shape = [n_thresholds]*) – A count of false positives, at index i being the number of negative samples assigned a score >= thresholds[i]. The total number of negative samples is equal to fps[-1] (thus true negatives are given by fps[-1] - fps).
- **tps** (*array, shape = [n_thresholds <= len(np.unique(y_score))]*) – An increasing count of true positives, at index i being the number of positive samples assigned a score >= thresholds[i]. The total number of positive samples is equal to tps[-1] (thus false negatives are given by tps[-1] - tps).
- **thresholds** (*array, shape = [n_thresholds]*) – Decreasing score values.

Example

```
>>> y_true  = [      1,   1,   1,   1,   1,   1,   0]
>>> y_score = [ np.nan, 0.2, 0.3, 0.4, 0.5, 0.6, 0.3]
>>> sample_weight = None
>>> pos_label = None
>>> fps, tps, thresholds = _binary_clf_curve2(y_true, y_score)
```

kwcoco.metrics.util

Module Contents

Classes

DictProxy	Allows an object to proxy the behavior of a dict attribute

class kwcoco.metrics.util.**DictProxy**

Bases: scriptconfig.dict_like.DictLike

Allows an object to proxy the behavior of a dict attribute

__getitem__(*self*, *key*)

__setitem__(*self*, *key*, *value*)

keys(*self*)

__json__(*self*)

kwcoco.metrics.voc_metrics

Module Contents

Classes

VOC_Metrics	API to compute object detection scores using Pascal VOC evaluation method.

Functions

_pr_curves(y, method='voc2012')	Compute a PR curve from a method
_voc_eval(lines, recs, classname, iou_thresh=0.5, method='voc2012', bias=1.0)	VOC AP evaluation for a single category.
_voc_ave_precision(rec, prec, method='voc2012')	Compute AP from precision and recall

class kwcoco.metrics.voc_metrics.**VOC_Metrics**(*classes=None*)

Bases: ubelt.NiceRepr

API to compute object detection scores using Pascal VOC evaluation method.

To use, add true and predicted detections for each image and then run the *score* function.

Variables

- **recs** (*Dict[int, List[dict]]*) – true boxes for each image. maps image ids to a list of records within that image. Each record is a tlbr bbox, a difficult flag, and a class name.
- **cx_to_lines** (*Dict[int, List]*) – VOC formatted prediction preditions. mapping from class index to all predictions for that category. Each "line" is a list of [

  [<imgid>, <score>, <tl_x>, <tl_y>, <br_x>, <br_y>]].

__nice__(*self*)

add_truth(*self*, *true_dets*, *gid*)

add_predictions(*self*, *pred_dets*, *gid*)

score(*self*, *iou_thresh=0.5*, *bias=1*, *method='voc2012'*)

Compute VOC scores for every category

Example

```
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> from kwcoco.metrics.voc_metrics import *  # NOQA
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=1, nboxes=(0, 100), n_fp=(0, 30), n_fn=(0, 30), classes=2, score_
↪noise=0.9)
>>> self = VOC_Metrics(classes=dmet.classes)
>>> self.add_truth(dmet.true_detections(0), 0)
>>> self.add_predictions(dmet.pred_detections(0), 0)
>>> voc_scores = self.score()
```

(continues on next page)

(continued from previous page)

```
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, doclf=True)
>>> voc_scores['perclass'].draw()
```

kwplot.figure(fnum=2) dmet.true_detections(0).draw(color='green', labels=None) dmet.pred_detections(0).draw(color='blue', labels=None) kwplot.autoplt().gca().set_xlim(0, 100) kwplot.autoplt().gca().set_ylim(0, 100)

kwcoco.metrics.voc_metrics.**_pr_curves**(*y*, *method='voc2012'*)

Compute a PR curve from a method

Parameters **y** (*pd.DataFrame | DataFrameArray*) – output of detection_confusions

Returns Tuple[float, ndarray, ndarray]

Example

```
>>> import pandas as pd
>>> y1 = pd.DataFrame.from_records([
>>>     {'pred': 0, 'score': 10.00, 'true': -1, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.65, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  8.64, 'true': -1, 'weight': 1.00},
>>>     {'pred': 0, 'score':  3.97, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.68, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  5.06, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  0.25, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  1.75, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  8.52, 'true':  0, 'weight': 1.00},
>>>     {'pred': 0, 'score':  5.20, 'true':  0, 'weight': 1.00},
>>> ])
>>> import kwarray
>>> y2 = kwarray.DataFrameArray(y1)
>>> _pr_curves(y2)
>>> _pr_curves(y1)
```

kwcoco.metrics.voc_metrics.**_voc_eval**(*lines*, *recs*, *classname*, *iou_thresh=0.5*, *method='voc2012'*, *bias=1.0*)

VOC AP evaluation for a single category.

Parameters

- **lines** (*List[list]*) – VOC formatted predictions. Each "line" is a list of [[<imgid>, <score>, <tl_x>, <tl_y>, <br_x>, <br_y>]].
- **recs** (*Dict[int, List[dict]*) – true boxes for each image. maps image ids to a list of records within that image. Each record is a tlbr bbox, a difficult flag, and a class name.
- **classname** (*str*) – the category to evaluate.
- **method** (*str*) – code for how the AP is computed.
- **bias** (*float*) – either 1.0 or 0.0.

Returns

info about the evaluation containing AP. Contains fp, tp, prec, rec,

Return type Dict

Notes

Raw replication of matlab implementation of creating assignments and the resulting PR-curves and AP. Based on MATLAB code [1].

References

[1] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCdevkit_18-May-2011.tar

kwcoco.metrics.voc_metrics.**_voc_ave_precision**(*rec*, *prec*, *method='voc2012'*)

Compute AP from precision and recall Based on MATLAB code in[1],[2], and[3].

Parameters

- **rec** (*ndarray*) – recall
- **prec** (*ndarray*) – precision
- **method** (*str*) – either voc2012 or voc2007

Returns ap: average precision

Return type float

References

Package Contents

Classes

DetectionMetrics	Object that computes associations between detections and can convert them
BinaryConfusionVectors	Stores information about a binary classification problem.
ConfusionVectors	Stores information used to construct a confusion matrix. This includes
Measures	**Example**
OneVsRestConfusionVectors	Container for multiple one-vs-rest binary confusion vectors
PerClass_Measures	

[1] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCdevkit_18-May-2011.tar
[2] https://github.com/rbgirshick/voc-dpm/blob/master/test/pascal_eval.m
[3] https://github.com/rbgirshick/voc-dpm/blob/c0b88564bd668bcc6216bbffe96cb061613be768/utils/bootstrap/VOCevaldet_bootstrap.m

Functions

eval_detections_cli(**kw)	DEPRECATED USE *kwcoco eval* instead

class kwcoco.metrics.**DetectionMetrics**(*dmet*, *classes=None*)

Bases: ubelt.NiceRepr

Object that computes associations between detections and can convert them into sklearn-compatible representations for scoring.

Variables

- **gid_to_true_dets** (*Dict*) – maps image ids to truth
- **gid_to_pred_dets** (*Dict*) – maps image ids to predictions
- **classes** (CategoryTree) – category coder

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), classes=8, score_noise=0.9,␣
↪hacked=False)
>>> print(dmet.score_kwcoco(bias=0, compat='mutex', prioritize='iou')['mAP'])
...
>>> # NOTE: IN GENERAL NETHARN AND VOC ARE NOT THE SAME
>>> print(dmet.score_voc(bias=0)['mAP'])
0.8582...
>>> #print(dmet.score_coco()['mAP'])
```

score_coco

clear(*dmet*)

__nice__(*dmet*)

classmethod from_coco(*DetectionMetrics*, *true_coco*, *pred_coco*, *gids=None*, *verbose=0*)

Create detection metrics from two coco files representing the truth and predictions.

Parameters

- **true_coco** (*kwcoco.CocoDataset*)
- **pred_coco** (*kwcoco.CocoDataset*)

Example

```
>>> import kwcoco
>>> from kwcoco.demo.perterb import perterb_coco
>>> true_coco = kwcoco.CocoDataset.demo('shapes')
>>> perterbkw = dict(box_noise=0.5, cls_noise=0.5, score_noise=0.5)
>>> pred_coco = perterb_coco(true_coco, **perterbkw)
>>> self = DetectionMetrics.from_coco(true_coco, pred_coco)
>>> self.score_voc()
```

_register_imagename(*dmet*, *imgname*, *gid=None*)

add_predictions(*dmet*, *pred_dets*, *imgname=None*, *gid=None*)
Register/Add predicted detections for an image

Parameters

- **pred_dets** (*Detections*) – predicted detections
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int, optional*) – the integer image id if known

add_truth(*dmet*, *true_dets*, *imgname=None*, *gid=None*)
Register/Add groundtruth detections for an image

Parameters

- **true_dets** (*Detections*) – groundtruth
- **imgname** (*str*) – a unique string to identify the image
- **gid** (*int, optional*) – the integer image id if known

true_detections(*dmet*, *gid*)
gets Detections representation for groundtruth in an image

pred_detections(*dmet*, *gid*)
gets Detections representation for predictions in an image

confusion_vectors(*dmet*, *iou_thresh=0.5*, *bias=0*, *gids=None*, *compat='mutex'*, *prioritize='iou'*, *ignore_classes='ignore'*, *background_class=ub.NoParam*, *verbose='auto'*, *workers=0*, *track_probs='try'*, *max_dets=None*)
Assigns predicted boxes to the true boxes so we can transform the detection problem into a classification problem for scoring.

Parameters

- **iou_thresh** (*float | List[float], default=0.5*) – bounding box overlap iou threshold required for assignment if a list, then return type is a dict
- **bias** (*float, default=0.0*) – for computing bounding box overlap, either 1 or 0
- **gids** (*List[int], default=None*) – which subset of images ids to compute confusion metrics on. If not specified all images are used.
- **compat** (*str, default='all'*) – can be ('ancestors' | 'mutex' | 'all'). determines which pred boxes are allowed to match which true boxes. If 'mutex', then pred boxes can only match true boxes of the same class. If 'ancestors', then pred boxes can match true boxes that match or have a coarser label. If 'all', then any pred can match any true, regardless of its category label.
- **prioritize** (*str, default='iou'*) – can be ('iou' | 'class' | 'correct') determines which box to assign to if mutiple true boxes overlap a predicted box. if prioritize is iou, then the true box with maximum iou (above iou_thresh) will be chosen. If prioritize is class, then it will prefer matching a compatible class above a higher iou. If prioritize is correct, then ancestors of the true class are preferred over descendents of the true class, over unreleated classes.
- **ignore_classes** (*set, default={'ignore'}*) – class names indicating ignore regions
- **background_class** (*str, default=ub.NoParam*) – Name of the background class. If unspecified we try to determine it with heuristics. A value of None means there is no background class.
- **verbose** (*int, default='auto'*) – verbosity flag. In auto mode, verbose=1 if len(gids) > 1000.

- **workers** (*int, default=0*) – number of parallel assignment processes
- **track_probs** (*str, default='try'*) – can be 'try', 'force', or False. if truthy, we assume probabilities for multiple classes are available.

Returns ConfusionVectors | Dict[float, ConfusionVectors]

Ignore: globals().update(xdev.get_func_kwargs(dmet.confusion_vectors))

Example

```
>>> dmet = DetectionMetrics.demo(nimgs=30, classes=3,
>>>                              nboxes=10, n_fp=3, box_noise=10,
>>>                              with_probs=False)
>>> iou_to_cfsn = dmet.confusion_vectors(iou_thresh=[0.3, 0.5, 0.9])
>>> for t, cfsn in iou_to_cfsn.items():
>>>     print('t = {!r}'.format(t))
...     print(cfsn.binarize_ovr().measures())
...     print(cfsn.binarize_classless().measures())
```

score_kwant(*dmet*, *iou_thresh=0.5*)
Scores the detections using kwant

score_kwcoco(*dmet*, *iou_thresh=0.5*, *bias=0*, *gids=None*, *compat='all'*, *prioritize='iou'*)
our scoring method

score_voc(*dmet*, *iou_thresh=0.5*, *bias=1*, *method='voc2012'*, *gids=None*, *ignore_classes='ignore'*)
score using voc method

Example

```
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=100, nboxes=(0, 3), n_fp=(0, 1), classes=8,
>>>     score_noise=.5)
>>> print(dmet.score_voc()['mAP'])
0.9399...
```

_to_coco(*dmet*)
Convert to a coco representation of truth and predictions

with inverse aid mappings

score_pycocotools(*dmet*, *with_evaler=False*, *with_confusion=False*, *verbose=0*, *iou_thresholds=None*)
score using ms-coco method

Returns dictionary with pct info

Return type Dict

Example

```
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> from kwcoco.metrics.detect_metrics import *
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 3), n_fn=(0, 1), n_fp=(0, 1), classes=8, with_
↪probs=False)
>>> pct_info = dmet.score_pycocotools(verbose=1,
>>>                                   with_evaler=True,
>>>                                   with_confusion=True,
>>>                                   iou_thresholds=[0.5, 0.9])
>>> evaler = pct_info['evaler']
>>> iou_to_cfsn_vecs = pct_info['iou_to_cfsn_vecs']
>>> for iou_thresh in iou_to_cfsn_vecs.keys():
>>>     print('iou_thresh = {!r}'.format(iou_thresh))
>>>     cfsn_vecs = iou_to_cfsn_vecs[iou_thresh]
>>>     ovr_measures = cfsn_vecs.binarize_ovr().measures()
>>>     print('ovr_measures = {}'.format(ub.repr2(ovr_measures, nl=1,␣
↪precision=4)))
```

Notes

by default pycocotools computes average precision as the literal average of computed precisions at 101 uniformly spaced recall thresholds.

pycocoutils seems to only allow predictions with the same category as the truth to match those truth objects. This should be the same as calling dmet.confusion_vectors with compat = mutex

pycocoutils does not take into account the fact that each box often has a score for each category.

pycocoutils will be incorrect if any annotation has an id of 0

a major difference in the way kwcoco scores versus pycocoutils is the calculation of AP. The assignment between truth and predicted detections produces similar enough results. Given our confusion vectors we use the scikit-learn definition of AP, whereas pycocoutils seems to compute precision and recall — more or less correctly — but then it resamples the precision at various specified recall thresholds (in the *accumulate* function, specifically how *pr* is resampled into the *q* array). This can lead to a large difference in reported scores.

pycocoutils also smooths out the precision such that it is monotonic decreasing, which might not be the best idea.

pycocotools area ranges are inclusive on both ends, that means the "small" and "medium" truth selections do overlap somewhat.

classmethod demo(*cls*, ***kwargs*)

Creates random true boxes and predicted boxes that have some noisy offset from the truth.

Kwargs:

classes (int, default=1): class list or the number of foreground classes.

nimgs (int, default=1): number of images in the coco datasts. nboxes (int, default=1): boxes per image. n_fp (int, default=0): number of false positives. n_fn (int, default=0): number of false negatives. box_noise (float, default=0): std of a normal distribution used to

perterb both box location and box size.

cls_noise (float, default=0): probability that a class label will change. Must be within 0 and 1.

anchors (ndarray, default=None): used to create random boxes null_pred (bool, default=0):

if True, predicted classes are returned as null, which means only localization scoring is suitable.

with_probs (bool, default=1): if True, includes per-class probabilities with predictions

CommandLine: xdoctest -m kwcoco.metrics.detect_metrics DetectionMetrics.demo:2 –show

Example

```
>>> kwargs = {}
>>> # Seed the RNG
>>> kwargs['rng'] = 0
>>> # Size parameters determine how big the data is
>>> kwargs['nimgs'] = 5
>>> kwargs['nboxes'] = 7
>>> kwargs['classes'] = 11
>>> # Noise parameters perterb predictions further from the truth
>>> kwargs['n_fp'] = 3
>>> kwargs['box_noise'] = 0.1
>>> kwargs['cls_noise'] = 0.5
>>> dmet = DetectionMetrics.demo(**kwargs)
>>> print('dmet.classes = {}'.format(dmet.classes))
dmet.classes = <CategoryTree(nNodes=12, maxDepth=3, maxBreadth=4...)>
>>> # Can grab kwimage.Detection object for any image
>>> print(dmet.true_detections(gid=0))
<Detections(4)>
>>> print(dmet.pred_detections(gid=0))
<Detections(7)>
```

Example

```
>>> # Test case with null predicted categories
>>> dmet = DetectionMetrics.demo(nimgs=30, null_pred=1, classes=3,
>>>                              nboxes=10, n_fp=3, box_noise=0.1,
>>>                              with_probs=False)
>>> dmet.gid_to_pred_dets[0].data
>>> dmet.gid_to_true_dets[0].data
>>> cfsn_vecs = dmet.confusion_vectors()
>>> binvecs_ovr = cfsn_vecs.binarize_ovr()
>>> binvecs_per = cfsn_vecs.binarize_classless()
>>> measures_per = binvecs_per.measures()
>>> measures_ovr = binvecs_ovr.measures()
>>> print('measures_per = {!r}'.format(measures_per))
>>> print('measures_ovr = {!r}'.format(measures_ovr))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> measures_ovr['perclass'].draw(key='pr', fnum=2)
```

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 1), nimgs=32, nboxes=(0, 16),
>>>     classes=3, rng=0, newstyle=1, box_noise=0.5, cls_noise=0.0, score_
↪noise=0.3, with_probs=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> summary = dmet.summarize(plot=True, title='DetectionMetrics summary demo',␣
↪with_ovr=True, with_bin=False)
>>> summary['bin_measures']
>>> kwplot.show_if_requested()
```

summarize(*dmet*, *out_dpath=None*, *plot=False*, *title=''*, *with_bin='auto'*, *with_ovr='auto'*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 128), n_fn=(0, 4), nimgs=512, nboxes=(0, 32),
>>>     classes=3, rng=0)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> dmet.summarize(plot=True, title='DetectionMetrics summary demo')
>>> kwplot.show_if_requested()
```

kwcoco.metrics.**eval_detections_cli**(***kw*)

DEPRECATED USE *kwcoco eval* instead

CommandLine: xdoctest -m ~/code/kwcoco/kwcoco/metrics/detect_metrics.py eval_detections_cli

class kwcoco.metrics.**BinaryConfusionVectors**(*data*, *cx=None*, *classes=None*)

Bases: ubelt.NiceRepr

Stores information about a binary classification problem. This is always with respect to a specific class, which is given by *cx* and *classes*.

The *data* DataFrameArray must contain *is_true* - if the row is an instance of class *classes[cx]* *pred_score* - the predicted probability of class *classes[cx]*, and *weight* - sample weight of the example

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=10)
>>> print('self = {!r}'.format(self))
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=0)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=1)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=2)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

classmethod demo(*cls*, *n=10*, *p_true=0.5*, *p_error=0.2*, *p_miss=0.0*, *rng=None*)

Create random data for tests

Parameters

- **n** (*int*) – number of rows
- **p_true** (*int*) – fraction of real positive cases
- **p_error** (*int*) – probability of making a recoverable mistake
- **p_miss** (*int*) – probability of making a unrecoverable mistake
- **rng** (*int | RandomState*) – random seed / state

Returns BinaryConfusionVectors

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn = BinaryConfusionVectors.demo(n=1000, p_error=0.1, p_miss=0.1)
>>> measures = cfsn.measures()
>>> print('measures = {}'.format(ub.repr2(measures, nl=1)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=1, pnum=(1, 2, 1))
>>> measures.draw('pr')
>>> kwplot.figure(fnum=1, pnum=(1, 2, 2))
>>> measures.draw('roc')
```

property catname(*self*)

__nice__(*self*)

__len__(*self*)

measures(*self*, *stabalize_thresh=7*, *fp_cutoff=None*, *monotonic_ppv=True*, *ap_method='pycocotools'*)

Get statistics (F1, G1, MCC) versus thresholds

Parameters

- **stabalize_thresh** (*int, default=7*) – if fewer than this many data points inserts dummy stabalization data so curves can still be drawn.
- **fp_cutoff** (*int, default=None*) – maximum number of false positives in the truncated roc curves. `None` is equivalent to `float('inf')`
- **monotonic_ppv** (*bool, default=True*) – if True ensures that precision is always increasing as recall decreases. This is done in pycocotools scoring, but I'm not sure its a good idea.

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=0)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=1, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> self = BinaryConfusionVectors.demo(n=3, p_true=0.5, p_error=0.5)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
```

```
>>> self = BinaryConfusionVectors.demo(n=100, p_true=0.5, p_error=0.5, p_miss=0.
↪3)
>>> print('measures = {}'.format(ub.repr2(self.measures())))
>>> print('measures = {}'.format(ub.repr2(ub.odict(self.measures()))))
```

Ignore:

import matplotlib.cm as cm # kwargs = {} # cmap = kwargs.get('cmap', mpl.cm.coolwarm) # n = len(x) # xgrid = np.tile(x[None, :], (n, 1)) # ygrid = np.tile(y[None, :], (n, 1)) # zdata = np.tile(z[None, :], (n, 1)) # ax.contour(xgrid, ygrid, zdata, zdir='x', cmap=cmap) # ax.contour(xgrid, ygrid, zdata, zdir='y', cmap=cmap) # ax.contour(xgrid, ygrid, zdata, zdir='z', cmap=cmap)

References

https://en.wikipedia.org/wiki/Confusion_matrix https://en.wikipedia.org/wiki/Precision_and_recall https://en.wikipedia.org/wiki/Matthews_correlation_coefficient

Ignore: self.measures().summary_plot() import xdev globals().update(xdev.get_func_kwargs(BinaryConfusionVectors.mea

_binary_clf_curves(*self*, *stabalize_thresh=7*, *fp_cutoff=None*)

Compute TP, FP, TN, and FN counts for this binary confusion vector.

Code common to ROC, PR, and threshold measures, computes the elements of the binary confusion matrix at all relevant operating point thresholds.

Parameters

- **stabalize_thresh** (*int*) – if fewer than this many data points insert stabalization data.
- **fp_cutoff** (*int*) – maximum number of false positives

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> self = BinaryConfusionVectors.demo(n=1, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

```
>>> self = BinaryConfusionVectors.demo(n=0, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

```
>>> self = BinaryConfusionVectors.demo(n=100, p_true=0.5, p_error=0.5)
>>> self._binary_clf_curves()
```

Ignore: import xdev globals().update(xdev.get_func_kwargs(BinaryConfusionVectors._binary_clf_curves)) >>> self = BinaryConfusionVectors.demo(n=10, p_true=0.7, p_error=0.3, p_miss=0.2) >>> print('measures = {}'.format(ub.repr2(self._binary_clf_curves()))) >>> info = self.measures() >>> info = ub.dict_isect(info, ['tpr', 'fpr', 'ppv', 'fp_count']) >>> print('measures = {}'.format(ub.repr2(ub.odict(i))))

draw_distribution(*self*)

_3dplot(*self*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 2), nimgs=256, nboxes=(0, 10),
>>>     bbox_noise=10,
>>>     classes=1)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> self = bin_cfsn = cfsn_vecs.binarize_classless()
>>> #dmet.summarize(plot=True)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.figure(fnum=3)
>>> self._3dplot()
```

class kwcoco.metrics.**ConfusionVectors**(*cfsn_vecs*, *data*, *classes*, *probs=None*)

Bases: ubelt.NiceRepr

Stores information used to construct a confusion matrix. This includes corresponding vectors of predicted labels, true labels, sample weights, etc...

Variables

- **data** (*DataFrameArray*) – should at least have keys true, pred, weight
- **classes** (*Sequence | CategoryTree*) – list of category names or category graph
- **probs** (*ndarray, optional*) – probabilities for each class

Example

```
>>> # xdoctest: IGNORE_WANT
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> print(cfsn_vecs.data._pandas())
     pred  true   score  weight     iou  txs  pxs  gid
0       2     2 10.0000  1.0000  1.0000    0    4    0
1       2     2  7.5025  1.0000  1.0000    1    3    0
2       1     1  5.0050  1.0000  1.0000    2    2    0
3       3    -1  2.5075  1.0000 -1.0000   -1    1    0
4       2    -1  0.0100  1.0000 -1.0000   -1    0    0
5      -1     2  0.0000  1.0000 -1.0000    3   -1    0
6      -1     2  0.0000  1.0000 -1.0000    4   -1    0
7       2     2 10.0000  1.0000  1.0000    0    5    1
8       2     2  8.0020  1.0000  1.0000    1    4    1
9       1     1  6.0040  1.0000  1.0000    2    3    1
..    ...   ...     ...     ...     ...  ...  ...  ...
62     -1     2  0.0000  1.0000 -1.0000    7   -1    7
63     -1     3  0.0000  1.0000 -1.0000    8   -1    7
64     -1     1  0.0000  1.0000 -1.0000    9   -1    7
65      1    -1 10.0000  1.0000 -1.0000   -1    0    8
66      1     1  0.0100  1.0000  1.0000    0    1    8
67      3    -1 10.0000  1.0000 -1.0000   -1    3    9
68      2     2  6.6700  1.0000  1.0000    0    2    9
69      2     2  3.3400  1.0000  1.0000    1    1    9
70      3    -1  0.0100  1.0000 -1.0000   -1    0    9
71     -1     2  0.0000  1.0000 -1.0000    2   -1    9
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco.metrics.confusion_vectors import ConfusionVectors
>>> cfsn_vecs = ConfusionVectors.demo(
>>>     nimgs=128, nboxes=(0, 10), n_fp=(0, 3), n_fn=(0, 3), classes=3)
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> measures = cx_to_binvecs.measures()['perclass']
>>> print('measures = {!r}'.format(measures))
measures = <PerClass_Measures({
    'cat_1': <Measures({'ap': 0.227, 'auc': 0.507, 'catname': cat_1, 'max_f1': f1=0.
↪45@0.47, 'nsupport': 788.000})>,
    'cat_2': <Measures({'ap': 0.288, 'auc': 0.572, 'catname': cat_2, 'max_f1': f1=0.
↪51@0.43, 'nsupport': 788.000})>,
    'cat_3': <Measures({'ap': 0.225, 'auc': 0.484, 'catname': cat_3, 'max_f1': f1=0.
↪46@0.40, 'nsupport': 788.000})>,
}) at 0x7facf77bdfd0>
>>> kwplot.figure(fnum=1, doclf=True)
>>> measures.draw(key='pr', fnum=1, pnum=(1, 3, 1))
>>> measures.draw(key='roc', fnum=1, pnum=(1, 3, 2))
```

(continues on next page)

(continued from previous page)

```
>>> measures.draw(key='mcc', fnum=1, pnum=(1, 3, 3))
...
```

__nice__(*cfsn_vecs*)

__json__(*self*)
Serialize to json

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import ConfusionVectors
>>> self = ConfusionVectors.demo(n_imgs=1, classes=2, n_fp=0, nboxes=1)
>>> state = self.__json__()
>>> print('state = {}'.format(ub.repr2(state, nl=2, precision=2, align=1)))
>>> recon = ConfusionVectors.from_json(state)
```

classmethod from_json(*cls*, *state*)

classmethod demo(*cfsn_vecs*, ***kw*)

Parameters ****kwargs** – See *kwcoco.metrics.DetectionMetrics.demo()*

Returns ConfusionVectors

Example

```
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> cx_to_binvecs = cfsn_vecs.binarize_ovr()
>>> print('cx_to_binvecs = {!r}'.format(cx_to_binvecs))
```

classmethod from_arrays(*ConfusionVectors*, *true*, *pred=None*, *score=None*, *weight=None*, *probs=None*, *classes=None*)
Construct confusion vector data structure from component arrays

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> import kwarray
>>> classes = ['person', 'vehicle', 'object']
>>> rng = kwarray.ensure_rng(0)
>>> true = (rng.rand(10) * len(classes)).astype(int)
>>> probs = rng.rand(len(true), len(classes))
>>> cfsn_vecs = ConfusionVectors.from_arrays(true=true, probs=probs,␣
↪classes=classes)
>>> cfsn_vecs.confusion_matrix()
pred     person  vehicle  object
real
person        0        0       0
```

(continues on next page)

(continued from previous page)

```
vehicle       2        4       1
object        2        1       0
```

confusion_matrix(*cfsn_vecs*, *compress=False*)

Builds a confusion matrix from the confusion vectors.

Parameters **compress** (*bool, default=False*) – if True removes rows / columns with no entries

Returns

cm [the labeled confusion matrix]

(Note: we should write a efficient replacement for this use case. #remove_pandas)

Return type pd.DataFrame

CommandLine: xdoctest -m kwcoco.metrics.confusion_vectors ConfusionVectors.confusion_matrix

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), n_fn=(0, 1), classes=3, cls_
↪noise=.2)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> cm = cfsn_vecs.confusion_matrix()
...
>>> print(cm.to_string(float_format=lambda x: '%.2f' % x))
pred        background  cat_1  cat_2  cat_3
real
background        0.00   1.00   2.00   3.00
cat_1             3.00  12.00   0.00   0.00
cat_2             3.00   0.00  14.00   0.00
cat_3             2.00   0.00   0.00  17.00
```

coarsen(*cfsn_vecs*, *cxs*)

Creates a coarsened set of vectors

Returns ConfusionVectors

binarize_classless(*cfsn_vecs*, *negative_classes=None*)

Creates a binary representation useful for measuring the performance of detectors. It is assumed that scores of "positive" classes should be high and "negative" clases should be low.

Parameters **negative_classes** (*List[str | int]*) – list of negative class names or idxs, by default chooses any class with a true class index of -1. These classes should ideally have low scores.

Returns BinaryConfusionVectors

Notes

The "classlessness" of this depends on the compat="all" argument being used when constructing confusion vectors, otherwise it becomes something like a macro-average because the class information was used in deciding which true and predicted boxes were allowed to match.

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), n_fn=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> class_idxs = list(dmet.classes.node_to_idx.values())
>>> binvecs = cfsn_vecs.binarize_classless()
```

binarize_ovr(*cfsn_vecs*, *mode=1*, *keyby='name'*, *ignore_classes={'ignore'}*, *approx=0*)

Transforms cfsn_vecs into one-vs-rest BinaryConfusionVectors for each category.

Parameters

- **mode** (*int, default=1*) – 0 for heirarchy aware or 1 for voc like. MODE 0 IS PROBABLY BROKEN
- **keyby** (*int | str*) – can be cx or name
- **ignore_classes** (*Set[str]*) – category names to ignore
- **approx** (*bool, default=0*) – if True try and approximate missing scores otherwise assume they are irrecoverable and use -inf

Returns

which behaves like Dict[int, BinaryConfusionVectors]: cx_to_binvecs

Return type *OneVsRestConfusionVectors*

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> print('cfsn_vecs = {!r}'.format(cfsn_vecs))
>>> catname_to_binvecs = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('catname_to_binvecs = {!r}'.format(catname_to_binvecs))
```

cfsn_vecs.data.pandas() catname_to_binvecs.cx_to_binvecs['class_1'].data.pandas()

Notes

classification_report(*cfsn_vecs*, *verbose=0*)
Build a classification report with various metrics.

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo()
>>> report = cfsn_vecs.classification_report(verbose=1)
```

class kwcoco.metrics.**Measures**(*info*)
Bases: ubelt.NiceRepr, *kwcoco.metrics.util.DictProxy*

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> binvecs = BinaryConfusionVectors.demo(n=100, p_error=0.5)
>>> self = binvecs.measures()
>>> print('self = {!r}'.format(self))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.draw(doclf=True)
>>> self.draw(key='pr',  pnum=(1, 2, 1))
>>> self.draw(key='roc', pnum=(1, 2, 2))
>>> kwplot.show_if_requested()
```

property catname(*self*)

__nice__(*self*)

reconstruct(*self*)

classmethod from_json(*cls*, *state*)

__json__(*self*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> binvecs = BinaryConfusionVectors.demo(n=10, p_error=0.5)
>>> self = binvecs.measures()
>>> info = self.__json__()
>>> print('info = {}'.format(ub.repr2(info, nl=1)))
>>> populate_info(info)
>>> print('info = {}'.format(ub.repr2(info, nl=1)))
>>> recon = Measures.from_json(info)
```

summary(*self*)

draw(*self*, *key=None*, *prefix=''*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(module:pandas)
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

summary_plot(*self*, *fnum=1*, *title=''*, *subplots='auto'*)

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> cfsn_vecs = ConfusionVectors.demo(n=3, p_error=0.5)
>>> binvecs = cfsn_vecs.binarize_classless()
>>> self = binvecs.measures()
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> self.summary_plot()
>>> kwplot.show_if_requested()
```

class kwcoco.metrics.**OneVsRestConfusionVectors**(*cx_to_binvecs*, *classes*)

Bases: ubelt.NiceRepr

Container for multiple one-vs-rest binary confusion vectors

Variables

- **cx_to_binvecs** –
- **classes** –

Example

```
>>> from kwcoco.metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     nimgs=10, nboxes=(0, 10), n_fp=(0, 1), classes=3)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> self = cfsn_vecs.binarize_ovr(keyby='name')
>>> print('self = {!r}'.format(self))
```

__nice__(*self*)

classmethod demo(*cls*)

Parameters ****kwargs** – See *kwcoco.metrics.DetectionMetrics.demo()*

Returns ConfusionVectors

keys(*self*)

__getitem__(*self*, *cx*)

measures(*self*, *stabalize_thresh=7*, *fp_cutoff=None*, *monotonic_ppv=True*, *ap_method='pycocotools'*)
Creates binary confusion measures for every one-versus-rest category.

Parameters

- **stabalize_thresh** (*int, default=7*) – if fewer than this many data points inserts dummy stabalization data so curves can still be drawn.
- **fp_cutoff** (*int, default=None*) – maximum number of false positives in the truncated roc curves. `None` is equivalent to `float('inf')`
- **monotonic_ppv** (*bool, default=True*) – if True ensures that precision is always increasing as recall decreases. This is done in pycocotools scoring, but I'm not sure its a good idea.

SeeAlso: *BinaryConfusionVectors.measures()*

Example

```
>>> self = OneVsRestConfusionVectors.demo()
>>> thresh_result = self.measures()['perclass']
```

abstract ovr_classification_report(*self*)

class kwcoco.metrics.**PerClass_Measures**(*cx_to_info*)
Bases: `ubelt.NiceRepr`, *kwcoco.metrics.util.DictProxy*

__nice__(*self*)

summary(*self*)

classmethod from_json(*cls*, *state*)

__json__(*self*)

draw(*self*, *key='mcc'*, *prefix=''*, ***kw*)

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> cfsn_vecs = ConfusionVectors.demo()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name')
>>> self = ovr_cfsn.measures()['perclass']
>>> self.draw('mcc', doclf=True, fnum=1)
>>> self.draw('pr', doclf=1, fnum=2)
>>> self.draw('roc', doclf=1, fnum=3)
```

draw_roc(*self*, *prefix=''*, ***kw*)

draw_pr(*self*, *prefix=''*, ***kw*)

summary_plot(*self*, *fnum=1*, *title=''*, *subplots='auto'*)

CommandLine: python ~/code/kwcoco/kwcoco/metrics/confusion_vectors.py PerClass_Measures.summary_plot –show

Example

```
>>> from kwcoco.metrics.confusion_vectors import *  # NOQA
>>> from kwcoco.metrics.detect_metrics import DetectionMetrics
>>> dmet = DetectionMetrics.demo(
>>>     n_fp=(0, 1), n_fn=(0, 3), nimgs=32, nboxes=(0, 32),
>>>     classes=3, rng=0, newstyle=1, box_noise=0.7, cls_noise=0.2, score_
↪noise=0.3, with_probs=False)
>>> cfsn_vecs = dmet.confusion_vectors()
>>> ovr_cfsn = cfsn_vecs.binarize_ovr(keyby='name', ignore_classes=['vector',
↪'raster'])
>>> self = ovr_cfsn.measures()['perclass']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> import seaborn as sns
>>> sns.set()
>>> self.summary_plot(title='demo summary_plot ovr', subplots=['pr', 'roc'])
>>> kwplot.show_if_requested()
>>> self.summary_plot(title='demo summary_plot ovr', subplots=['mcc', 'acc'],␣
↪fnum=2)
```

2.2.6 kwcoco.util

mkinit ~/code/kwcoco/kwcoco/util/__init__.py -w

Submodules

kwcoco.util.dict_like

Module Contents

Classes

DictLike	An inherited class must specify the `getitem`, `setitem`, and

class kwcoco.util.dict_like.**DictLike**

Bases: ubelt.NiceRepr

An inherited class must specify the `getitem`, `setitem`, and `keys` methods.

A class is dictionary like if it has:

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

`__iter__`, `__len__`, `__contains__`, `__getitem__`, `items`, `keys`, `values`, `get`,

and if it should be writable it should have: `__delitem__`, `__setitem__`, `update`,

And perhaps: `copy`,

asdict

abstract getitem(*self*, *key*)

abstract setitem(*self*, *key*, *value*)

abstract delitem(*self*, *key*)

abstract keys(*self*)

__len__(*self*)

__iter__(*self*)

__contains__(*self*, *key*)

__delitem__(*self*, *key*)

__getitem__(*self*, *key*)

__setitem__(*self*, *key*, *value*)

items(*self*)

values(*self*)

copy(*self*)

to_dict(*self*)

update(*self*, *other*)

iteritems(*self*)

itervalues(*self*)

iterkeys(*self*)

get(*self*, *key*, *default=None*)

kwcoco.util.jsonschema_elements

Functional interface into defining jsonschema structures.

See mixin classes for details.

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem = SchemaElements()
>>> for base in SchemaElements.__bases__:
>>>     print('\n\n====\nbase = {!r}'.format(base))
>>>     attrs = [key for key in dir(base) if not key.startswith('_')]
>>>     for key in attrs:
>>>         value = getattr(elem, key)
>>>         print('{} = {}'.format(key, value))
```

Module Contents

Classes

Element	A dictionary used to define an element of a JSON Schema.
ScalarElements	Single-valued elements
QuantifierElements	Quantifier types
ContainerElements	Types that contain other types
SchemaElements	Functional interface into defining jsonschema structures.

Attributes

elem
ALLOF
ANY
ANYOF
ARRAY
BOOLEAN
INTEGER
NOT
NULL
NUMBER
OBJECT

continues on next page

Table 63 – continued from previous page

ONEOF
STRING

class kwcoco.util.jsonschema_elements.**Element**(*base*, *options={}*, *_magic=None*)

Bases: dict

A dictionary used to define an element of a JSON Schema.

The exact keys/values for the element will depend on the type of element being described. The *SchemaElements* defines exactly what these are for the core elements. (e.g. OBJECT, INTEGER, NULL, ARRAY, ANYOF)

Example

```
>>> from kwcoco.coco_schema import *  # NOQA
>>> self = Element(base={'type': 'demo'}, options={'opt1', 'opt2'})
>>> new = self(opt1=3)
>>> print('self = {}'.format(ub.repr2(self, nl=1)))
>>> print('new = {}'.format(ub.repr2(new, nl=1)))
>>> print('new2 = {}'.format(ub.repr2(new(), nl=1)))
>>> print('new3 = {}'.format(ub.repr2(new(title='myvar'), nl=1)))
>>> print('new4 = {}'.format(ub.repr2(new(title='myvar')(examples=['']), nl=1)))
>>> print('new5 = {}'.format(ub.repr2(new(badattr=True), nl=1)))
self = {
    'type': 'demo',
}
new = {
    'opt1': 3,
    'type': 'demo',
}
new2 = {
    'opt1': 3,
    'type': 'demo',
}
new3 = {
    'opt1': 3,
    'title': 'myvar',
    'type': 'demo',
}
new4 = {
    'examples': [''],
    'opt1': 3,
    'title': 'myvar',
    'type': 'demo',
}
new5 = {
    'opt1': 3,
    'type': 'demo',
}
```

__generics__

__call__(*self*, **args*, ***kw*)

validate(*self*, *instance=ub.NoParam*)

If `instance` is given, validates that that dictionary conforms to this schema. Otherwise validates that this is a valid schema element.

Parameters **instance** (*dict*) – a dictionary to validate

__or__(*self*, *other*)

Syntax for making an anyOf relationship

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> obj1 = OBJECT(dict(opt1=NUMBER()))
>>> obj2 = OBJECT(dict(opt2=STRING()))
>>> obj3 = OBJECT(dict(opt3=ANY()))
>>> any_v1 = obj1 | obj2
>>> any_v2 = ANYOF(obj1, obj2)
>>> assert any_v1 == any_v2
>>> any_v3 = any_v1 | obj3
>>> any_v4 = ANYOF(obj1, obj2, obj3)
>>> assert any_v3 == any_v4
```

class kwcoco.util.jsonschema_elements.**ScalarElements**

Bases: object

Single-valued elements

property **NULL**(*self*)

https://json-schema.org/understanding-json-schema/reference/null.html

property **BOOLEAN**(*self*)

https://json-schema.org/understanding-json-schema/reference/null.html

property **STRING**(*self*)

https://json-schema.org/understanding-json-schema/reference/string.html

property **NUMBER**(*self*)

https://json-schema.org/understanding-json-schema/reference/numeric.html#number

property **INTEGER**(*self*)

https://json-schema.org/understanding-json-schema/reference/numeric.html#integer

class kwcoco.util.jsonschema_elements.**QuantifierElements**

Bases: object

Quantifier types

https://json-schema.org/understanding-json-schema/reference/combining.html#allof

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem.ANYOF(elem.STRING, elem.NUMBER).validate()
>>> elem.ONEOF(elem.STRING, elem.NUMBER).validate()
>>> elem.NOT(elem.NULL).validate()
>>> elem.NOT(elem.ANY).validate()
>>> elem.ANY.validate()
```

property ANY(*self*)

ALLOF(*self*, **TYPES*)

ANYOF(*self*, **TYPES*)

ONEOF(*self*, **TYPES*)

NOT(*self*, *TYPE*)

class kwcoco.util.jsonschema_elements.**ContainerElements**

Types that contain other types

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> print(elem.ARRAY().validate())
>>> print(elem.OBJECT().validate())
>>> print(elem.OBJECT().validate())
{'type': 'array', 'items': {}}
{'type': 'object', 'properties': {}}
{'type': 'object', 'properties': {}}
```

ARRAY(*self*, *TYPE={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/array.html

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> ARRAY(numItems=3)
>>> schema = ARRAY(minItems=3)
>>> schema.validate()
{'type': 'array', 'items': {}, 'minItems': 3}
```

OBJECT(*self*, *PROPERTIES={}*, ***kw*)

https://json-schema.org/understanding-json-schema/reference/object.html

Example

```
>>> import jsonschema
>>> schema = elem.OBJECT()
>>> jsonschema.validate({}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.ANY(),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'])
>>> jsonschema.validate({'key1': None}, schema)
>>> #
>>> import jsonschema
>>> schema = elem.OBJECT({
>>>     'key1': elem.OBJECT({'arr': elem.ARRAY()}),
>>>     'key2': elem.ANY(),
>>> }, required=['key1'], title='a title')
>>> schema.validate()
>>> print('schema = {}'.format(ub.repr2(schema, nl=-1)))
>>> jsonschema.validate({'key1': {'arr': []}}, schema)
schema = {
    'properties': {
        'key1': {
            'properties': {
                'arr': {'items': {}, 'type': 'array'}
            },
            'type': 'object'
        },
        'key2': {}
    },
    'required': ['key1'],
    'title': 'a title',
    'type': 'object'
}
```

class kwcoco.util.jsonschema_elements.**SchemaElements**

Bases: *ScalarElements*, *QuantifierElements*, *ContainerElements*

Functional interface into defining jsonschema structures.

See mixin classes for details.

References

https://json-schema.org/understanding-json-schema/

Todo:

- [ ] Generics: title, description, default, examples

CommandLine: xdoctest -m /home/joncrall/code/kwcoco/kwcoco/util/jsonschema_elements.py SchemaElements

Example

```
>>> from kwcoco.util.jsonschema_elements import *  # NOQA
>>> elem = SchemaElements()
>>> elem.ARRAY(elem.ANY())
>>> schema = OBJECT({
>>>     'prop1': ARRAY(INTEGER, minItems=3),
>>>     'prop2': ARRAY(STRING, numItems=2),
>>>     'prop3': ARRAY(OBJECT({
>>>         'subprob1': NUMBER,
>>>         'subprob2': NUMBER,
>>>     }))
>>> })
>>> print('schema = {}'.format(ub.repr2(schema, nl=2)))
schema = {
    'properties': {
        'prop1': {'items': {'type': 'integer'}, 'minItems': 3, 'type': 'array'},
        'prop2': {'items': {'type': 'string'}, 'maxItems': 2, 'minItems': 2, 'type
↪': 'array'},
        'prop3': {'items': {'properties': {'subprob1': {'type': 'number'}, 'subprob2
↪': {'type': 'number'}}, 'type': 'object'}, 'type': 'array'},
    },
    'type': 'object',
}
```

```
>>> TYPE = elem.OBJECT({
>>>     'p1': ANY,
>>>     'p2': ANY,
>>> }, required=['p1'])
>>> import jsonschema
>>> inst = {'p1': None}
>>> jsonschema.validate(inst, schema=TYPE)
>>> #jsonschema.validate({'p2': None}, schema=TYPE)
```

kwcoco.util.jsonschema_elements.**elem**

kwcoco.util.jsonschema_elements.**ALLOF**

kwcoco.util.jsonschema_elements.**ANY**

kwcoco.util.jsonschema_elements.**ANYOF**

kwcoco.util.jsonschema_elements.**ARRAY**

kwcoco.util.jsonschema_elements.**BOOLEAN**

kwcoco.util.jsonschema_elements.**INTEGER**

kwcoco.util.jsonschema_elements.**NOT**

kwcoco.util.jsonschema_elements.**NULL**

kwcoco.util.jsonschema_elements.**NUMBER**

kwcoco.util.jsonschema_elements.**OBJECT**

kwcoco.util.jsonschema_elements.**ONEOF**

kwcoco.util.jsonschema_elements.**STRING**

kwcoco.util.util_delayed_poc

This module is ported from ndsampler, and will likely eventually move to kwimage and be refactored using pymbolic

The classes in this file represent a tree of delayed operations.

Proof of concept for delayed chainable transforms in Python.

There are several optimizations that could be applied.

This is similar to GDAL's virtual raster table, but it works in memory and I think it is easier to chain operations.

SeeAlso: ../../dev/symbolic_delayed.py

Warning: As the name implies this is a proof of concept, and the actual implementation was hacked together too quickly. Serious refactoring will be necessary.

Concepts:

> Each class should be a layer that adds a new transformation on top of underlying nested layers. Adding new layers should be quick, and there should always be the option to "finalize" a stack of layers, chaining the transforms / operations and then applying one final efficient transform at the end.

Conventions:

- dsize = (always in width / height), no channels are present
- shape for images is always (height, width, channels)
- channels are always the last dimension of each image, if no channel dim is specified, finalize will add it.
- **Videos must be the last process in the stack, and add a leading** time dimension to the shape. dsize is still width, height, but shape is now: (time, height, width, chan)

Example

```
>>> # Example demonstrating the modivating use case
>>> # We have multiple aligned frames for a video, but each of
>>> # those frames is in a different resolution. Furthermore,
>>> # each of the frames consists of channels in different resolutions.
>>> # Create raw channels in some "native" resolution for frame 1
>>> f1_chan1 = DelayedIdentity.demo('astro', chan=0, dsize=(300, 300))
>>> f1_chan2 = DelayedIdentity.demo('astro', chan=1, dsize=(200, 200))
>>> f1_chan3 = DelayedIdentity.demo('astro', chan=2, dsize=(10, 10))
>>> # Create raw channels in some "native" resolution for frame 2
>>> f2_chan1 = DelayedIdentity.demo('carl', dsize=(64, 64), chan=0)
>>> f2_chan2 = DelayedIdentity.demo('carl', dsize=(260, 260), chan=1)
>>> f2_chan3 = DelayedIdentity.demo('carl', dsize=(10, 10), chan=2)
>>> #
>>> # Delayed warp each channel into its "image" space
>>> # Note: the images never actually enter this space we transform through it
>>> f1_dsize = np.array((3, 3))
>>> f2_dsize = np.array((2, 2))
>>> f1_img = DelayedChannelConcat([
>>>     f1_chan1.delayed_warp(Affine.scale(f1_dsize / f1_chan1.dsize), dsize=f1_dsize),
>>>     f1_chan2.delayed_warp(Affine.scale(f1_dsize / f1_chan2.dsize), dsize=f1_dsize),
```

(continues on next page)

(continued from previous page)

```
>>>     f1_chan3.delayed_warp(Affine.scale(f1_dsize / f1_chan3.dsize), dsize=f1_dsize),
>>> ])
>>> f2_img = DelayedChannelConcat([
>>>     f2_chan1.delayed_warp(Affine.scale(f2_dsize / f2_chan1.dsize), dsize=f2_dsize),
>>>     f2_chan2.delayed_warp(Affine.scale(f2_dsize / f2_chan2.dsize), dsize=f2_dsize),
>>>     f2_chan3.delayed_warp(Affine.scale(f2_dsize / f2_chan3.dsize), dsize=f2_dsize),
>>> ])
>>> # Combine frames into a video
>>> vid_dsize = np.array((280, 280))
>>> vid = DelayedFrameConcat([
>>>     f1_img.delayed_warp(Affine.scale(vid_dsize / f1_img.dsize), dsize=vid_dsize),
>>>     f2_img.delayed_warp(Affine.scale(vid_dsize / f2_img.dsize), dsize=vid_dsize),
>>> ])
>>> vid.nesting
>>> print('vid.nesting = {}'.format(ub.repr2(vid.__json__(), nl=-2)))
>>> final = vid.finalize(interpolation='nearest')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final[0], pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(final[1], pnum=(1, 2, 2), fnum=1)
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.delayed_load(1)
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> astro = DelayedLoad.demo('astro')
>>> print('MSI = ' + ub.repr2(delayed.__json__(), nl=-3, sort=0))
>>> print('ASTRO = ' + ub.repr2(astro.__json__(), nl=2, sort=0))
```

```
>>> subchan = delayed.take_channels('B1|B8')
>>> subcrop = subchan.delayed_crop((slice(10, 80), slice(30, 50)))
>>> #
>>> subcrop.nesting()
>>> subchan.nesting()
>>> subchan.finalize()
>>> subcrop.finalize()
>>> #
>>> msi_crop = delayed.delayed_crop((slice(10, 80), slice(30, 50)))
>>> subdata = msi_crop.delayed_warp(kwimage.Affine.scale(3), dsize='auto').take_channels(
↪'B11|B1')
>>> subdata.finalize()
```

Module Contents

Classes

DelayedVisionOperation	Base class for nodes in a tree of delayed computer-vision operations
DelayedVideoOperation	Base class for nodes in a tree of delayed computer-vision operations
DelayedImageOperation	Operations that pertain only to images
DelayedIdentity	Noop leaf that does nothing. Mostly used in tests atm.
DelayedNans	Constructs nan channels as needed
DelayedLoad	A load operation for a specific sub-region and sub-bands in a specified
LazyGDalFrameFile	
DelayedFrameConcat	Represents multiple frames in a video
DelayedChannelConcat	Represents multiple channels in an image that could be concatenated
DelayedWarp	POC for chainable transforms
DelayedCrop	Represent a delayed crop operation

Functions

have_gdal()	
validate_nonzero_data(file)	Test to see if the image is all black.
_rectify_slice_dim(part, D)	
_compute_leaf_subcrop(root_region_bounds, tf_leaf_to_root)	Given a region in a "root" image and a trasnform between that "root" and
_largest_shape(shapes)	Finds maximum over all shapes
_devcheck_corner()	

Attributes

profile	
_GDAL_DTYPE_LUT	

kwcoco.util.util_delayed_poc.**profile**

class kwcoco.util.util_delayed_poc.**DelayedVisionOperation**

Bases: ubelt.NiceRepr

Base class for nodes in a tree of delayed computer-vision operations

__nice__(*self*)

abstract finalize(*self*)

abstract children(*self*)
Abstract method, which should generate all of the direct children of a node in the operation tree.

_optimize_paths(*self*, ***kwargs*)
Iterate through the leaf nodes, which are virtually transformed into the root space.

This returns some sort of hueristically optimized leaf repr wrt warps.

__json__(*self*)

nesting(*self*)

warp(*self*, **args*, ***kwargs*)
alias for delayed_warp, might change to this API in the future

crop(*self*, **args*, ***kwargs*)
alias for delayed_crop, might change to this API in the future

class kwcoco.util.util_delayed_poc.**DelayedVideoOperation**
Bases: *DelayedVisionOperation*

Base class for nodes in a tree of delayed computer-vision operations

class kwcoco.util.util_delayed_poc.**DelayedImageOperation**
Bases: *DelayedVisionOperation*

Operations that pertain only to images

delayed_crop(*self*, *region_slices*)
Create a new delayed image that performs a crop in the transformed "self" space.

Parameters **region_slices** (*Tuple[slice, slice]*) – y-slice and x-slice.

Notes

Returns a heuristically "simplified" tree. In the current implementation there are only 3 operations, cat, warp, and crop. All cats go at the top, all crops go at the bottom, all warps are in the middle.

Returns lazy executed delayed transform

Return type *DelayedWarp*

Example

```
>>> dsize = (100, 100)
>>> tf2 = Affine.affine(scale=3).matrix
>>> self = DelayedWarp(np.random.rand(33, 33), tf2, dsize)
>>> region_slices = (slice(5, 10), slice(1, 12))
>>> delayed_crop = self.delayed_crop(region_slices)
>>> print(ub.repr2(delayed_crop.nesting(), nl=-1, sort=0))
>>> delayed_crop.finalize()
```

Example

```
>>> chan1 = DelayedLoad.demo('astro')
>>> chan2 = DelayedLoad.demo('carl')
>>> warped1a = chan1.delayed_warp(Affine.scale(1.2).matrix)
>>> warped2a = chan2.delayed_warp(Affine.scale(1.5))
>>> warped1b = warped1a.delayed_warp(Affine.scale(1.2).matrix)
>>> warped2b = warped2a.delayed_warp(Affine.scale(1.5))
>>> #
>>> region_slices = (slice(97, 677), slice(5, 691))
>>> self = warped2b
>>> #
>>> crop1 = warped1b.delayed_crop(region_slices)
>>> crop2 = warped2b.delayed_crop(region_slices)
>>> print(ub.repr2(warped1b.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(warped2b.nesting(), nl=-1, sort=0))
>>> # Notice how the crop merges the two nesting layers
>>> # (via the hueristic optimize step)
>>> print(ub.repr2(crop1.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(crop2.nesting(), nl=-1, sort=0))
>>> frame1 = crop1.finalize(dsize=(500, 500))
>>> frame2 = crop2.finalize(dsize=(500, 500))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(frame1, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(frame2, pnum=(1, 2, 2), fnum=1)
```

delayed_warp(*self*, *transform*, *dsize=None*)

Delayedly transform the underlying data.

Note: this deviates from kwimage warp functions because instead of “output_dims” (specified in c-style shape) we specify dsize (w, h).

Returns new delayed transform a chained transform

Return type *DelayedWarp*

abstract take_channels(*self*, *channels*)

class kwcoco.util.util_delayed_poc.**DelayedIdentity**(*sub_data*)

Bases: *DelayedImageOperation*

Noop leaf that does nothing. Mostly used in tests atm.

Typically used to just hold raw data.

DelayedIdentity.demo(‘astro’, chan=0, dsize=(32, 32))

__hack_dont_optimize__ = True

classmethod demo(*cls*, *key='astro'*, *chan=None*, *dsize=None*)

children(*self*)

Abstract method, which should generate all of the direct children of a node in the operation tree.

_optimize_paths(*self*, ***kwargs*)
Iterate through the leaf nodes, which are virtually transformed into the root space.

This returns some sort of hueristically optimized leaf repr wrt warps.

finalize(*self*)

class kwcoco.util.util_delayed_poc.**DelayedNans**(*dsize=None*, *channels=None*)
Bases: *DelayedImageOperation*

Constructs nan channels as needed

Example

self = DelayedNans((10, 10), channel_spec.FusedChannelSpec.coerce('rgb')) region_slices = (slice(5, 10), slice(1, 12)) delayed = self.delayed_crop(region_slices)

property shape(*self*)

property num_bands(*self*)

property dsize(*self*)

property channels(*self*)

children(*self*)
Abstract method, which should generate all of the direct children of a node in the operation tree.

_optimize_paths(*self*, ***kwargs*)
Iterate through the leaf nodes, which are virtually transformed into the root space.

This returns some sort of hueristically optimized leaf repr wrt warps.

finalize(*self*, ***kwargs*)

delayed_crop(*self*, *region_slices*)
Create a new delayed image that performs a crop in the transformed "self" space.

Parameters **region_slices** (*Tuple[slice, slice]*) – y-slice and x-slice.

Notes

Returns a heuristically "simplified" tree. In the current implementation there are only 3 operations, cat, warp, and crop. All cats go at the top, all crops go at the bottom, all warps are in the middle.

Returns lazy executed delayed transform

Return type *DelayedWarp*

Example

```
>>> dsize = (100, 100)
>>> tf2 = Affine.affine(scale=3).matrix
>>> self = DelayedWarp(np.random.rand(33, 33), tf2, dsize)
>>> region_slices = (slice(5, 10), slice(1, 12))
>>> delayed_crop = self.delayed_crop(region_slices)
>>> print(ub.repr2(delayed_crop.nesting(), nl=-1, sort=0))
>>> delayed_crop.finalize()
```

Example

```
>>> chan1 = DelayedLoad.demo('astro')
>>> chan2 = DelayedLoad.demo('carl')
>>> warped1a = chan1.delayed_warp(Affine.scale(1.2).matrix)
>>> warped2a = chan2.delayed_warp(Affine.scale(1.5))
>>> warped1b = warped1a.delayed_warp(Affine.scale(1.2).matrix)
>>> warped2b = warped2a.delayed_warp(Affine.scale(1.5))
>>> #
>>> region_slices = (slice(97, 677), slice(5, 691))
>>> self = warped2b
>>> #
>>> crop1 = warped1b.delayed_crop(region_slices)
>>> crop2 = warped2b.delayed_crop(region_slices)
>>> print(ub.repr2(warped1b.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(warped2b.nesting(), nl=-1, sort=0))
>>> # Notice how the crop merges the two nesting layers
>>> # (via the hueristic optimize step)
>>> print(ub.repr2(crop1.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(crop2.nesting(), nl=-1, sort=0))
>>> frame1 = crop1.finalize(dsize=(500, 500))
>>> frame2 = crop2.finalize(dsize=(500, 500))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(frame1, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(frame2, pnum=(1, 2, 2), fnum=1)
```

delayed_warp(*self*, *transform*, *dsize=None*)

Delayedly transform the underlying data.

Note: this deviates from kwimage warp functions because instead of “output_dims” (specified in c-style shape) we specify dsize (w, h).

Returns new delayed transform a chained transform

Return type *DelayedWarp*

class kwcoco.util.util_delayed_poc.**DelayedLoad**(*fpath*, *channels=None*, *dsize=None*, *num_bands=None*, *immediate_crop=None*, *immediate_chan_idxs=None*, *immediate_dsize=None*)

Bases: *DelayedImageOperation*

A load operation for a specific sub-region and sub-bands in a specified image.

Notes

This class contains support for fusing certain lazy operations into this layer, namely cropping, scaling, and channel selection.

For now these are named `immediates`

Example

```
>>> fpath = kwimage.grab_test_image_fpath()
>>> self = DelayedLoad(fpath)
>>> print('self = {!r}'.format(self))
>>> self.load_shape()
>>> print('self = {!r}'.format(self))
>>> self.finalize()
```

```
>>> f1_img = DelayedLoad.demo('astro', dsize=(300, 300))
>>> f2_img = DelayedLoad.demo('carl', dsize=(256, 320))
>>> print('f1_img = {!r}'.format(f1_img))
>>> print('f2_img = {!r}'.format(f2_img))
>>> print(f2_img.finalize().shape)
>>> print(f1_img.finalize().shape)
```

```
>>> fpath = kwimage.grab_test_image_fpath()
>>> channels = channel_spec.FusedChannelSpec.coerce('rgb')
>>> self = DelayedLoad(fpath, channels=channels)
```

__hack_dont_optimize__ = True

classmethod demo(*DelayedLoad*, *key='astro'*, *dsize=None*)

abstract classmethod coerce(*cls*, *data*)

children(*self*)
Abstract method, which should generate all of the direct children of a node in the operation tree.

nesting(*self*)

_optimize_paths(*self*, ***kwargs*)
Iterate through the leaf nodes, which are virtually transformed into the root space.

This returns some sort of hueristically optimized leaf repr wrt warps.

load_shape(*self*, *use_channel_heuristic=False*)

_ensure_dsize(*self*)

property shape(*self*)

property num_bands(*self*)

property dsize(*self*)

property channels(*self*)

property fpath(*self*)

finalize(*self*, ***kwargs*)

delayed_crop(*self*, *region_slices*)

Parameters **region_slices** (*Tuple[slice, slice]*) – y-slice and x-slice.

Returns a new delayed load object with a fused crop operation

Return type *DelayedLoad*

Example

```
>>> # Test chained crop operations
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> self = orig = DelayedLoad.demo('astro').load_shape()
>>> region_slices = slices1 = (slice(0, 90), slice(30, 60))
>>> self = crop1 = orig.delayed_crop(slices1)
>>> region_slices = slices2 = (slice(10, 21), slice(10, 22))
>>> self = crop2 = crop1.delayed_crop(slices2)
>>> region_slices = slices3 = (slice(3, 20), slice(5, 20))
>>> crop3 = crop2.delayed_crop(slices3)
>>> # Spot check internals
>>> print('orig = {}'.format(ub.repr2(orig.__json__(), nl=2)))
>>> print('crop1 = {}'.format(ub.repr2(crop1.__json__(), nl=2)))
>>> print('crop2 = {}'.format(ub.repr2(crop2.__json__(), nl=2)))
>>> print('crop3 = {}'.format(ub.repr2(crop3.__json__(), nl=2)))
>>> # Test internals
>>> assert crop3._immediates['crop'][0].start == 13
>>> assert crop3._immediates['crop'][0].stop == 21
>>> # Test shapes work out correctly
>>> assert crop3.finalize().shape == (8, 7, 3)
>>> assert crop2.finalize().shape == (11, 12, 3)
>>> assert crop1.take_channels([1, 2]).finalize().shape == (90, 30, 2)
>>> assert orig.finalize().shape == (512, 512, 3)
```

Notes

This chart gives an intuition on how new absolute slice coords are computed from existing absolute coords ane relative coords.

5 7 <- new 3 5 <- rel

2 9 <- curr

take_channels(*self*, *channels*)

This method returns a subset of the vision data with only the specified bands / channels.

Parameters **channels** (*List[int] | slice | channel_spec.FusedChannelSpec*) – List of integers indexes, a slice, or a channel spec, which is typically a pipe (|) delimited list of channel codes. See kwcoco.ChannelSpec for more detials.

Returns a new delayed load with a fused take channel operation

Return type *DelayedLoad*

Example

```
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> import kwcoco
>>> self = DelayedLoad.demo('astro').load_shape()
>>> channels = [2, 0]
>>> new = self.take_channels(channels)
>>> new3 = new.take_channels([1, 0])
```

```
>>> final1 = self.finalize()
>>> final2 = new.finalize()
>>> final3 = new3.finalize()
>>> assert np.all(final1[..., 2] == final2[..., 0])
>>> assert np.all(final1[..., 0] == final2[..., 1])
>>> assert final2.shape[2] == 2
```

```
>>> assert np.all(final1[..., 2] == final3[..., 1])
>>> assert np.all(final1[..., 0] == final3[..., 0])
>>> assert final3.shape[2] == 2
```

kwcoco.util.util_delayed_poc.**have_gdal**()

kwcoco.util.util_delayed_poc.**_GDAL_DTYPE_LUT**

class kwcoco.util.util_delayed_poc.**LazyGDalFrameFile**(*fpath*)

Bases: ubelt.NiceRepr

Todo:

- [ ] Move to its own backend module
- **[ ] When used with COCO, allow the image metadata to populate the** height, width, and channels if possible.

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> self = LazyGDalFrameFile.demo()
>>> print('self = {!r}'.format(self))
>>> self[0:3, 0:3]
>>> self[:, :, 0]
>>> self[0]
>>> self[0, 3]
```

```
>>> # import kwplot
>>> # kwplot.imshow(self[:])
```

_ds(*self*)

classmethod demo(*cls*, *key='astro'*, *dsize=None*)

Ignore:

```
>>> self = LazyGDalFrameFile.demo(dsize=(6600, 4400))
```

property ndim(*self*)

property shape(*self*)

property dtype(*self*)

__nice__(*self*)

__getitem__(*self*, *index*)

References

https://gis.stackexchange.com/questions/162095/gdal-driver-create-typeerror

Ignore:

```
>>> self = LazyGDalFrameFile.demo(dsize=(6600, 4400))
>>> index = [slice(2100, 2508, None), slice(4916, 5324, None), None]
>>> img_part = self[index]
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img_part)
```

__array__(*self*)
Allow this object to be passed to np.asarray

References

https://numpy.org/doc/stable/user/basics.dispatch.html

kwcoco.util.util_delayed_poc.**validate_nonzero_data**(*file*)
Test to see if the image is all black.

May fail on all-black images

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> import kwimage
>>> gpath = kwimage.grab_test_image_fpath()
>>> file = LazyGDalFrameFile(gpath)
>>> validate_nonzero_data(file)
```

kwcoco.util.util_delayed_poc.**_rectify_slice_dim**(*part*, *D*)

class kwcoco.util.util_delayed_poc.**DelayedFrameConcat**(*frames*, *dsize=None*)
Bases: *DelayedVideoOperation*

Represents multiple frames in a video

Notes

Video[0]:

Frame[0]: Chan[0]: (32) +————————————+ Chan[1]: (16) +————————+ Chan[2]: (8) +———+

Frame[1]: Chan[0]: (30) +————————————+ Chan[1]: (14) +————————+ Chan[2]: (6) +———+

Todo:

- [ ] Support computing the transforms when none of the data is loaded

---

Example

```
>>> # Simpler case with fewer nesting levels
>>> rng = kwarray.ensure_rng(None)
>>> # Delayed warp each channel into its "image" space
>>> # Note: the images never enter the space we transform through
>>> f1_img = DelayedLoad.demo('astro', (300, 300))
>>> f2_img = DelayedLoad.demo('carl', (256, 256))
>>> # Combine frames into a video
>>> vid_dsize = np.array((100, 100))
>>> self = vid = DelayedFrameConcat([
>>>     f1_img.delayed_warp(Affine.scale(vid_dsize / f1_img.dsize)),
>>>     f2_img.delayed_warp(Affine.scale(vid_dsize / f2_img.dsize)),
>>> ], dsize=vid_dsize)
>>> print(ub.repr2(vid.nesting(), nl=-1, sort=0))
>>> final = vid.finalize(interpolation='nearest', dsize=(32, 32))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final[0], pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(final[1], pnum=(1, 2, 2), fnum=1)
>>> region_slices = (slice(0, 90), slice(30, 60))
```

children(*self*)
Abstract method, which should generate all of the direct children of a node in the operation tree.

property channels(*self*)

property shape(*self*)

finalize(*self*, ***kwargs*)
Execute the final transform

delayed_crop(*self*, *region_slices*)

Example

```
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> # Create raw channels in some "native" resolution for frame 1
>>> f1_chan1 = DelayedIdentity.demo('astro', chan=(1, 0), dsize=(300, 300))
>>> f1_chan2 = DelayedIdentity.demo('astro', chan=2, dsize=(10, 10))
>>> # Create raw channels in some "native" resolution for frame 2
>>> f2_chan1 = DelayedIdentity.demo('carl', dsize=(64, 64), chan=(1, 0))
>>> f2_chan2 = DelayedIdentity.demo('carl', dsize=(10, 10), chan=2)
>>> #
>>> f1_dsize = np.array(f1_chan1.dsize)
>>> f2_dsize = np.array(f2_chan1.dsize)
>>> f1_img = DelayedChannelConcat([
>>>     f1_chan1.delayed_warp(Affine.scale(f1_dsize / f1_chan1.dsize), dsize=f1_
↪dsize),
>>>     f1_chan2.delayed_warp(Affine.scale(f1_dsize / f1_chan2.dsize), dsize=f1_
↪dsize),
>>> ])
>>> f2_img = DelayedChannelConcat([
>>>     f2_chan1.delayed_warp(Affine.scale(f2_dsize / f2_chan1.dsize), dsize=f2_
↪dsize),
>>>     f2_chan2.delayed_warp(Affine.scale(f2_dsize / f2_chan2.dsize), dsize=f2_
↪dsize),
>>> ])
>>> vid_dsize = np.array((280, 280))
>>> full_vid = DelayedFrameConcat([
>>>     f1_img.delayed_warp(Affine.scale(vid_dsize / f1_img.dsize), dsize=vid_
↪dsize),
>>>     f2_img.delayed_warp(Affine.scale(vid_dsize / f2_img.dsize), dsize=vid_
↪dsize),
>>> ])
>>> region_slices = (slice(80, 200), slice(80, 200))
>>> print(ub.repr2(full_vid.nesting(), nl=-1, sort=0))
>>> crop_vid = full_vid.delayed_crop(region_slices)
>>> final_full = full_vid.finalize(interpolation='nearest')
>>> final_crop = crop_vid.finalize(interpolation='nearest')
>>> import pytest
>>> with pytest.raises(ValueError):
>>>     # should not be able to crop a crop yet
>>>     crop_vid.delayed_crop(region_slices)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final_full[0], pnum=(2, 2, 1), fnum=1)
>>> kwplot.imshow(final_full[1], pnum=(2, 2, 2), fnum=1)
>>> kwplot.imshow(final_crop[0], pnum=(2, 2, 3), fnum=1)
>>> kwplot.imshow(final_crop[1], pnum=(2, 2, 4), fnum=1)
```

class kwcoco.util.util_delayed_poc.**DelayedChannelConcat**(*components*, *dsize=None*)

Bases: *DelayedImageOperation*

Represents multiple channels in an image that could be concatenated

Variables **components** (*List[DelayedWarp]*) – a list of stackable channels. Each component may be comprised of multiple channels.

Todo:

- [ ] can this be generalized into a delayed concat?
- [ ] can all concats be delayed until the very end?

Example

```
>>> comp1 = DelayedWarp(np.random.rand(11, 7))
>>> comp2 = DelayedWarp(np.random.rand(11, 7, 3))
>>> comp3 = DelayedWarp(
>>>     np.random.rand(3, 5, 2),
>>>     transform=Affine.affine(scale=(7/5, 11/3)).matrix,
>>>     dsize=(7, 11)
>>> )
>>> components = [comp1, comp2, comp3]
>>> chans = DelayedChannelConcat(components)
>>> final = chans.finalize()
>>> assert final.shape == chans.shape
>>> assert final.shape == (11, 7, 6)
```

```
>>> # We should be able to nest DelayedChannelConcat inside virutal images
>>> frame1 = DelayedWarp(
>>>     chans, transform=Affine.affine(scale=2.2).matrix,
>>>     dsize=(20, 26))
>>> frame2 = DelayedWarp(
>>>     np.random.rand(3, 3, 6), dsize=(20, 26))
>>> frame3 = DelayedWarp(
>>>     np.random.rand(3, 3, 6), dsize=(20, 26))
```

```
>>> print(ub.repr2(frame1.nesting(), nl=-1, sort=False))
>>> frame1.finalize()
>>> vid = DelayedFrameConcat([frame1, frame2, frame3])
>>> print(ub.repr2(vid.nesting(), nl=-1, sort=False))
```

children(*self*)

Abstract method, which should generate all of the direct children of a node in the operation tree.

classmethod random(*cls*, *num_parts=3*, *rng=None*)

Example

```
>>> self = DelayedChannelConcat.random()
>>> print('self = {!r}'.format(self))
>>> print(ub.repr2(self.nesting(), nl=-1, sort=0))
```

property channels(*self*)

property shape(*self*)

finalize(*self*, ***kwargs*)
Execute the final transform

take_channels(*self*, *channels*)
This method returns a subset of the vision data with only the specified bands / channels.

Parameters **channels** (*List[int] | slice | channel_spec.FusedChannelSpec*) – List of integers indexes, a slice, or a channel spec, which is typically a pipe (|) delimited list of channel codes. See kwcoco.ChannelSpec for more detials.

Returns a delayed vision operation that only operates on the following channels.

Return type *DelayedVisionOperation*

Example

```
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = delayed = dset.delayed_load(1)
>>> channels = 'B11|B8|B1|B10'
>>> new = self.take_channels(channels)
```

Example

```
>>> # Complex case
>>> import kwcoco
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = dset.delayed_load(1)
>>> astro = DelayedLoad.demo('astro').load_shape(use_channel_heuristic=True)
>>> aligned = astro.warp(kwimage.Affine.scale(600 / 512), dsize='auto')
>>> self = combo = DelayedChannelConcat(delayed.components + [aligned])
>>> channels = 'B1|r|B8|g'
>>> new = self.take_channels(channels)
>>> new_cropped = new.crop((slice(10, 200), slice(12, 350)))
>>> datas = new_cropped.finalize()
>>> vizable = kwimage.normalize_intensity(datas, axis=2)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> stacked = kwimage.stack_images(vizable.transpose(2, 0, 1))
>>> kwplot.imshow(stacked)
```

Example

```
>>> # Test case where requested channel does not exist
>>> import kwcoco
>>> from kwcoco.util.util_delayed_poc import *  # NOQA
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = dset.delayed_load(1)
>>> channels = 'B1|foobar|bazbiz|B8'
>>> new = self.take_channels(channels)
>>> new_cropped = new.crop((slice(10, 200), slice(12, 350)))
>>> fused = new_cropped.finalize()
>>> assert fused.shape == (190, 338, 4)
>>> assert np.all(np.isnan(fused[..., 1:3]))
>>> assert not np.any(np.isnan(fused[..., 0]))
>>> assert not np.any(np.isnan(fused[..., 3]))
```

class kwcoco.util.util_delayed_poc.**DelayedWarp**(*sub_data*, *transform=None*, *dsize=None*)

Bases: *DelayedImageOperation*

POC for chainable transforms

Notes

"sub" is used to refer to the underlying data in its native coordinates and resolution.

"self" is used to refer to the data in the transformed coordinates that are exposed by this class.

Variables

- **sub_data** (DelayedWarp | *ArrayLike*) – array-like image data at a naitive resolution
- **transform** (*Transform*) – transforms data from native "sub"-image-space to "self"-image-space.

Example

```
>>> dsize = (12, 12)
>>> tf1 = np.array([[2, 0, 0], [0, 2, 0], [0, 0, 1]])
>>> tf2 = np.array([[3, 0, 0], [0, 3, 0], [0, 0, 1]])
>>> tf3 = np.array([[4, 0, 0], [0, 4, 0], [0, 0, 1]])
>>> band1 = DelayedWarp(np.random.rand(6, 6), tf1, dsize)
>>> band2 = DelayedWarp(np.random.rand(4, 4), tf2, dsize)
>>> band3 = DelayedWarp(np.random.rand(3, 3), tf3, dsize)
>>> #
>>> # Execute a crop in a one-level transformed space
>>> region_slices = (slice(5, 10), slice(0, 12))
>>> delayed_crop = band2.delayed_crop(region_slices)
>>> final_crop = delayed_crop.finalize()
>>> #
>>> # Execute a crop in a nested transformed space
>>> tf4 = np.array([[1.5, 0, 0], [0, 1.5, 0], [0, 0, 1]])
>>> chained = DelayedWarp(band2, tf4, (18, 18))
>>> delayed_crop = chained.delayed_crop(region_slices)
>>> final_crop = delayed_crop.finalize()
```

(continues on next page)

(continued from previous page)

```
>>> #
>>> tf4 = np.array([[.5, 0, 0], [0, .5, 0], [0, 0, 1]])
>>> chained = DelayedWarp(band2, tf4, (6, 6))
>>> delayed_crop = chained.delayed_crop(region_slices)
>>> final_crop = delayed_crop.finalize()
>>> #
>>> region_slices = (slice(1, 5), slice(2, 4))
>>> delayed_crop = chained.delayed_crop(region_slices)
>>> final_crop = delayed_crop.finalize()
```

Example

```
>>> dsize = (17, 12)
>>> tf = np.array([[5.2, 0, 1.1], [0, 3.1, 2.2], [0, 0, 1]])
>>> self = DelayedWarp(np.random.rand(3, 5, 13), tf, dsize=dsize)
>>> self.finalize().shape
```

classmethod random(*cls*, *nesting=(2, 5)*, *rng=None*)

Example

```
>>> self = DelayedWarp.random(nesting=(4, 7))
>>> print('self = {!r}'.format(self))
>>> print(ub.repr2(self.nesting(), nl=-1, sort=0))
```

property channels(*self*)

children(*self*)

Abstract method, which should generate all of the direct children of a node in the operation tree.

property dsize(*self*)

property num_bands(*self*)

property shape(*self*)

_optimize_paths(*self*, ***kwargs*)

Example

```
>>> self = DelayedWarp.random()
>>> leafs = list(self._optimize_paths())
>>> print('leafs = {!r}'.format(leafs))
```

finalize(*self*, *transform=None*, *dsize=None*, *interpolation='linear'*, ***kwargs*)

Execute the final transform

Can pass a parent transform to augment this underlying transform.

Parameters

- **transform** (*Transform*) – an additional transform to perform
- **dsize** (*Tuple[int, int]*) – overrides destination canvas size

Example

```
>>> tf = np.array([[0.9, 0, 3.9], [0, 1.1, -.5], [0, 0, 1]])
>>> raw = kwimage.grab_test_image(dsize=(54, 65))
>>> raw = kwimage.ensure_float01(raw)
>>> # Test nested finalize
>>> layer1 = raw
>>> num = 10
>>> for _ in range(num):
...     layer1  = DelayedWarp(layer1, tf, dsize='auto')
>>> final1 = layer1.finalize()
>>> # Test non-nested finalize
>>> layer2 = list(layer1._optimize_paths())[0]
>>> final2 = layer2.finalize()
>>> #
>>> print(ub.repr2(layer1.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(layer2.nesting(), nl=-1, sort=0))
>>> print('final1 = {!r}'.format(final1))
>>> print('final2 = {!r}'.format(final2))
>>> print('final1.shape = {!r}'.format(final1.shape))
>>> print('final2.shape = {!r}'.format(final2.shape))
>>> assert np.allclose(final1, final2)
>>> #
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(raw, pnum=(1, 3, 1), fnum=1)
>>> kwplot.imshow(final1, pnum=(1, 3, 2), fnum=1)
>>> kwplot.imshow(final2, pnum=(1, 3, 3), fnum=1)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Test aliasing
>>> s = DelayedIdentity.demo()
>>> s = DelayedIdentity.demo('checkerboard')
>>> a = s.delayed_warp(Affine.scale(0.05), dsize='auto')
>>> b = s.delayed_warp(Affine.scale(3), dsize='auto')
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> # It looks like downsampling linear and area is the same
>>> # Does warpAffine have no alias handling?
>>> pnum_ = kwplot.PlotNums(nRows=2, nCols=4)
>>> kwplot.imshow(a.finalize(interpolation='area'), pnum=pnum_(), title=
↪'warpAffine area')
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(a.finalize(interpolation='linear'), pnum=pnum_(), title=
↪'warpAffine linear')
>>> kwplot.imshow(a.finalize(interpolation='nearest'), pnum=pnum_(), title=
↪'warpAffine nearest')
>>> kwplot.imshow(a.finalize(interpolation='nearest', antialias=False),␣
↪pnum=pnum_(), title='warpAffine nearest AA=0')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'area'), pnum=pnum_(), title='resize area')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'linear'), pnum=pnum_(), title='resize linear')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'nearest'), pnum=pnum_(), title='resize nearest')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'cubic'), pnum=pnum_(), title='resize cubic')
```

take_channels(*self*, *channels*)

class kwcoco.util.util_delayed_poc.**DelayedCrop**(*sub_data*, *sub_slices*)

Bases: *DelayedImageOperation*

Represent a delayed crop operation

Example

```
>>> sub_data = DelayedLoad.demo()
>>> sub_slices = (slice(5, 10), slice(1, 12))
>>> self = DelayedCrop(sub_data, sub_slices)
>>> print(ub.repr2(self.nesting(), nl=-1, sort=0))
>>> final = self.finalize()
>>> print('final.shape = {!r}'.format(final.shape))
```

Example

```
>>> sub_data = DelayedLoad.demo()
>>> sub_slices = (slice(5, 10), slice(1, 12))
>>> crop1 = DelayedCrop(sub_data, sub_slices)
>>> import pytest
>>> # Should only error while huristics are in use.
>>> with pytest.raises(ValueError):
>>>     crop2 = DelayedCrop(crop1, sub_slices)
```

__hack_dont_optimize__ = True

property channels(*self*)

children(*self*)

Abstract method, which should generate all of the direct children of a node in the operation tree.

finalize(*self*, ***kwargs*)

abstract _optimize_paths(*self*, ***kwargs*)

Iterate through the leaf nodes, which are virtually transformed into the root space.

This returns some sort of hueristically optimized leaf repr wrt warps.

kwcoco.util.util_delayed_poc.**_compute_leaf_subcrop**(*root_region_bounds*, *tf_leaf_to_root*)
Given a region in a "root" image and a trasnform between that "root" and some "leaf" image, compute the appropriate quantized region in the "leaf" image and the adjusted transformation between that root and leaf.

Example

```
>>> region_slices = (slice(33, 100), slice(22, 62))
>>> region_shape = (100, 100, 1)
>>> root_region_box = kwimage.Boxes.from_slice(region_slices, shape=region_shape)
>>> root_region_bounds = root_region_box.to_polygons()[0]
>>> tf_leaf_to_root = Affine.affine(scale=7).matrix
>>> slices, tf_new = _compute_leaf_subcrop(root_region_bounds, tf_leaf_to_root)
>>> print('tf_new =\n{!r}'.format(tf_new))
>>> print('slices = {!r}'.format(slices))
```

Ignore:

root_region_bounds = kwimage.Coords.random(4) tf_leaf_to_root = np.eye(3) tf_leaf_to_root[0, 2] = -1e-11

kwcoco.util.util_delayed_poc.**_largest_shape**(*shapes*)
Finds maximum over all shapes

Example

```
>>> shapes = [
>>>     (10, 20), None, (None, 30), (40, 50, 60, None), (100,)
>>> ]
>>> largest = _largest_shape(shapes)
>>> print('largest = {!r}'.format(largest))
>>> assert largest == (100, 50, 60, None)
```

kwcoco.util.util_delayed_poc.**_devcheck_corner**()

kwcoco.util.util_futures

Module Contents

Classes

SerialExecutor	Implements the concurrent.futures API around a single-threaded backend
Executor	Wrapper around a specific executor.

class kwcoco.util.util_futures.**SerialExecutor**
Bases: object

Implements the concurrent.futures API around a single-threaded backend

Example

```
>>> with SerialExecutor() as executor:
>>>     futures = []
>>>     for i in range(100):
>>>         f = executor.submit(lambda x: x + 1, i)
>>>         futures.append(f)
>>>     for f in concurrent.futures.as_completed(futures):
>>>         assert f.result() > 0
>>>     for i, f in enumerate(futures):
>>>         assert i + 1 == f.result()
```

__enter__(*self*)

__exit__(*self*, *ex_type*, *ex_value*, *tb*)

submit(*self*, *func*, **args*, ***kw*)

shutdown(*self*)

class kwcoco.util.util_futures.**Executor**(*mode='thread'*, *max_workers=0*)

Bases: object

Wrapper around a specific executor.

Abstracts Serial, Thread, and Process Executor via arguments.

Parameters

- **mode** (*str, default='thread'*) – either thread, serial, or process
- **max_workers** (*int, default=0*) – number of workers. If 0, serial is forced.

__enter__(*self*)

__exit__(*self*, *ex_type*, *ex_value*, *tb*)

submit(*self*, *func*, **args*, ***kw*)

shutdown(*self*)

kwcoco.util.util_json

Module Contents

Classes

IndexableWalker	Traverses through a nested tree-liked indexable structure.

Functions

ensure_json_serializable(dict_, normalize_containers=False, verbose=0)	Attempt to convert common types (e.g. numpy) into something json complient
find_json_unserializable(data, quickcheck=False)	Recurse through json datastructure and find any component that
indexable_allclose(dct1, dct2, return_info=False)	Walks through two nested data structures and ensures that everything is

kwcoco.util.util_json.**ensure_json_serializable**(*dict_*, *normalize_containers=False*, *verbose=0*)

Attempt to convert common types (e.g. numpy) into something json complient

Convert numpy and tuples into lists

Parameters **normalize_containers** (*bool, default=False*) – if True, normalizes dict containers to be standard python structures.

Example

```
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
>>> data['foo']['a'] = 1
>>> data['foo']['b'] = (1, np.array([1, 2, 3]), {3: np.int32(3), 4: np.float16(1.0)}
↪)
>>> dict_ = data
>>> print(ub.repr2(data, nl=-1))
>>> assert list(find_json_unserializable(data))
>>> result = ensure_json_serializable(data, normalize_containers=True)
>>> print(ub.repr2(result, nl=-1))
>>> assert not list(find_json_unserializable(result))
>>> assert type(result) is dict
```

kwcoco.util.util_json.**find_json_unserializable**(*data*, *quickcheck=False*)

Recurse through json datastructure and find any component that causes a serialization error. Record the location of these errors in the datastructure as we recurse through the call tree.

Parameters

- **data** (*object*) – data that should be json serializable
- **quickcheck** (*bool*) – if True, check the entire datastructure assuming its ok before doing the python-based recursive logic.

Returns

list of "bad part" dictionaries containing items 'value' - the value that caused the serialization error 'loc' - which contains a list of key/indexes that can be used

to lookup the location of the unserializable value. If the "loc" is a list, then it indicates a rare case where a key in a dictionary is causing the serialization error.

Return type List[Dict]

Example

```
>>> from kwcoco.util.util_json import *  # NOQA
>>> part = ub.ddict(lambda: int)
>>> part['foo'] = ub.ddict(lambda: int)
>>> part['bar'] = np.array([1, 2, 3])
>>> part['foo']['a'] = 1
>>> # Create a dictionary with two unserializable parts
>>> data = [1, 2, {'nest1': [2, part]}, {frozenset({'badkey'}): 3, 2: 4}]
>>> parts = list(find_json_unserializable(data))
>>> print('parts = {}'.format(ub.repr2(parts, nl=1)))
>>> # Check expected structure of bad parts
>>> assert len(parts) == 2
>>> part = parts[1]
>>> assert list(part['loc']) == [2, 'nest1', 1, 'bar']
>>> # We can use the "loc" to find the bad value
>>> for part in parts:
>>>     # "loc" is a list of directions containing which keys/indexes
>>>     # to traverse at each descent into the data structure.
>>>     directions = part['loc']
>>>     curr = data
>>>     special_flag = False
>>>     for key in directions:
>>>         if isinstance(key, list):
>>>             # special case for bad keys
>>>             special_flag = True
>>>             break
>>>         else:
>>>             # normal case for bad values
>>>             curr = curr[key]
>>>     if special_flag:
>>>         assert part['data'] in curr.keys()
>>>         assert part['data'] is key[1]
>>>     else:
>>>         assert part['data'] is curr
```

class kwcoco.util.util_json.**IndexableWalker**(*data*, *dict_cls=(dict,)*, *list_cls=(list, tuple)*)

Bases: collections.abc.Generator

Traverses through a nested tree-liked indexable structure.

Generates a path and value to each node in the structure. The path is a list of indexes which if applied in order will reach the value.

The `__setitem__` method can be used to modify a nested value based on the path returned by the generator.

When generating values, you can use “send” to prevent traversal of a particular branch.

Example

```
>>> # Create nested data
>>> import numpy as np
>>> data = ub.ddict(lambda: int)
>>> data['foo'] = ub.ddict(lambda: int)
>>> data['bar'] = np.array([1, 2, 3])
>>> data['foo']['a'] = 1
>>> data['foo']['b'] = np.array([1, 2, 3])
>>> data['foo']['c'] = [1, 2, 3]
>>> data['baz'] = 3
>>> print('data = {}'.format(ub.repr2(data, nl=True)))
>>> # We can walk through every node in the nested tree
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     if path[-1] == 'c':
>>>         # Use send to prevent traversing this branch
>>>         got = walker.send(False)
>>>         # We can modify the value based on the returned path
>>>         walker[path] = 'changed the value of c'
>>> print('data = {}'.format(ub.repr2(data, nl=True)))
>>> assert data['foo']['c'] == 'changed the value of c'
```

Example

```
>>> # Test sending false for every data item
>>> import numpy as np
>>> data = {1: 1}
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     print('walk path = {}'.format(ub.repr2(path, nl=0)))
>>>     walker.send(False)
>>> data = {}
>>> walker = IndexableWalker(data)
>>> for path, value in walker:
>>>     walker.send(False)
```

__iter__(*self*)

Iterates through the indexable `self.data`

Can send a False flag to prevent a branch from being traversed

Yields *Tuple[List, Any]* – path (List): list of index operations to arrive at the value value (object): the value at the path

__next__(*self*)

returns next item from this generator

send(*self*, *arg*)

send(arg) -> send 'arg' into generator, return next yielded value or raise StopIteration.

throw(*self*, *type=None*, *value=None*, *traceback=None*)

throw(typ[,val[,tb]]) -> raise exception in generator, return next yielded value or raise StopIteration.

__setitem__(*self*, *path*, *value*)
Set nested value by path

Parameters

- **path** (*List*) – list of indexes into the nested structure
- **value** (*object*) – new value

__getitem__(*self*, *path*)
Get nested value by path

Parameters **path** (*List*) – list of indexes into the nested structure

Returns value

__delitem__(*self*, *path*)
Remove nested value by path

Note: It can be dangerous to use this while iterating (because we may try to descend into a deleted location) or on leaf items that are list-like (because the indexes of all subsequent items will be modified).

Parameters **path** (*List*) – list of indexes into the nested structure. The item at the last index will be removed.

_walk(*self*, *data*, *prefix=[]*)
Defines the underlying generator used by IndexableWalker

Yields

***Tuple[List, object]* | *None* – path (List) - a "path" through the nested data structure**
value (object) - the value indexed by that "path".

Can also yield None in the case that *send* is called on the generator.

kwcoco.util.util_json.**indexable_allclose**(*dct1*, *dct2*, *return_info=False*)
Walks through two nested data structures and ensures that everything is roughly the same.

Parameters

- **dct1** – a nested indexable item
- **dct2** – a nested indexable item

Example

```
>>> from kwcoco.util.util_json import indexable_allclose
>>> dct1 = {
>>>     'foo': [1.222222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> dct2 = {
>>>     'foo': [1.22222, 1.333],
>>>     'bar': 1,
>>>     'baz': [],
>>> }
>>> assert indexable_allclose(dct1, dct2)
```

kwcoco.util.util_monkey

Module Contents

Classes

SupressPrint	Temporarily replace the print function in a module with a noop

class kwcoco.util.util_monkey.**SupressPrint**(**mods*, ***kw*)

Bases: object

Temporarily replace the print function in a module with a noop

Parameters

- ***mods** – the modules to disable print in
- ****kw** – only accepts "enabled" enabled (bool, default=True): enables or disables this context

__enter__(*self*)

__exit__(*self*, *a*, *b*, *c*)

kwcoco.util.util_sklearn

Extensions to sklearn constructs

Module Contents

Classes

StratifiedGroupKFold	Stratified K-Folds cross-validator with Grouping

class kwcoco.util.util_sklearn.**StratifiedGroupKFold**(*n_splits=3*, *shuffle=False*, *random_state=None*)

Bases: sklearn.model_selection._split._BaseKFold

Stratified K-Folds cross-validator with Grouping

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of GroupKFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

Read more in the User Guide.

Parameters **n_splits** (*int, default=3*) – Number of folds. Must be at least 2.

_make_test_folds(*self*, *X*, *y=None*, *groups=None*)

Parameters

- **X** (*ndarray*) – data
- **y** (*ndarray*) – labels

- **groups** (*ndarray*) – groupids for items. Items with the same groupid must be placed in the same group.

Returns test_folds

Return type list

Example

```
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> groups = [1, 1, 3, 4, 2, 2, 7, 8, 8]
>>> y      = [1, 1, 1, 1, 2, 2, 2, 3, 3]
>>> X = np.empty((len(y), 0))
>>> self = StratifiedGroupKFold(random_state=rng, shuffle=True)
>>> skf_list = list(self.split(X=X, y=y, groups=groups))
...
>>> import ubelt as ub
>>> print(ub.repr2(skf_list, nl=1, with_dtype=False))
[
    (np.array([2, 3, 4, 5, 6]), np.array([0, 1, 7, 8])),
    (np.array([0, 1, 2, 7, 8]), np.array([3, 4, 5, 6])),
    (np.array([0, 1, 3, 4, 5, 6, 7, 8]), np.array([2])),
]
```

_iter_test_masks(*self*, *X*, *y=None*, *groups=None*)
Generates boolean masks corresponding to test sets.

By default, delegates to _iter_test_indices(X, y, groups)

split(*self*, *X*, *y*, *groups=None*)
Generate indices to split data into training and test set.

Package Contents

Classes

Executor	Wrapper around a specific executor.
SerialExecutor	Implements the concurrent.futures API around a single-threaded backend
StratifiedGroupKFold	Stratified K-Folds cross-validator with Grouping

class kwcoco.util.**Executor**(*mode='thread'*, *max_workers=0*)
Bases: object

Wrapper around a specific executor.

Abstracts Serial, Thread, and Process Executor via arguments.

Parameters

- **mode** (*str, default='thread'*) – either thread, serial, or process
- **max_workers** (*int, default=0*) – number of workers. If 0, serial is forced.

__enter__(*self*)

__exit__(*self*, *ex_type*, *ex_value*, *tb*)

submit(*self*, *func*, **args*, ***kw*)

shutdown(*self*)

class kwcoco.util.SerialExecutor
Bases: object

Implements the concurrent.futures API around a single-threaded backend

Example

```
>>> with SerialExecutor() as executor:
>>>     futures = []
>>>     for i in range(100):
>>>         f = executor.submit(lambda x: x + 1, i)
>>>         futures.append(f)
>>>     for f in concurrent.futures.as_completed(futures):
>>>         assert f.result() > 0
>>>     for i, f in enumerate(futures):
>>>         assert i + 1 == f.result()
```

__enter__(*self*)

__exit__(*self*, *ex_type*, *ex_value*, *tb*)

submit(*self*, *func*, **args*, ***kw*)

shutdown(*self*)

class kwcoco.util.StratifiedGroupKFold(*n_splits=3*, *shuffle=False*, *random_state=None*)
Bases: sklearn.model_selection._split._BaseKFold

Stratified K-Folds cross-validator with Grouping

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of GroupKFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

Read more in the User Guide.

Parameters **n_splits** (*int, default=3*) – Number of folds. Must be at least 2.

_make_test_folds(*self*, *X*, *y=None*, *groups=None*)

Parameters

- **X** (*ndarray*) – data
- **y** (*ndarray*) – labels
- **groups** (*ndarray*) – groupids for items. Items with the same groupid must be placed in the same group.

Returns test_folds

Return type list

Example

```
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> groups = [1, 1, 3, 4, 2, 2, 7, 8, 8]
>>> y      = [1, 1, 1, 1, 2, 2, 2, 3, 3]
>>> X = np.empty((len(y), 0))
>>> self = StratifiedGroupKFold(random_state=rng, shuffle=True)
>>> skf_list = list(self.split(X=X, y=y, groups=groups))
...
>>> import ubelt as ub
>>> print(ub.repr2(skf_list, nl=1, with_dtype=False))
[
    (np.array([2, 3, 4, 5, 6]), np.array([0, 1, 7, 8])),
    (np.array([0, 1, 2, 7, 8]), np.array([3, 4, 5, 6])),
    (np.array([0, 1, 3, 4, 5, 6, 7, 8]), np.array([2])),
]
```

_iter_test_masks(*self*, *X*, *y=None*, *groups=None*)
Generates boolean masks corresponding to test sets.

By default, delegates to _iter_test_indices(X, y, groups)

split(*self*, *X*, *y*, *groups=None*)
Generate indices to split data into training and test set.

2.3 Submodules

2.3.1 kwcoco.__main__

2.3.2 kwcoco.abstract_coco_dataset

Module Contents

Classes

AbstractCocoDataset	This is a common base for all variants of the Coco Dataset

class kwcoco.abstract_coco_dataset.**AbstractCocoDataset**
Bases: abc.ABC

This is a common base for all variants of the Coco Dataset

At the time of writing there is kwcoco.CocoDataset (which is the dictionary-based backend), and the kwcoco.coco_sql_dataset.CocoSqlDataset, which is experimental.

2.3.3 kwcoco.category_tree

The category_tree` module defines the *CategoryTree* class, which is used for maintaining flat or hierarchical category information. The kwcoco version of this class only contains the datastructure and does not contain any torch operations. See the ndsampler version for the extension with torch operations.

Module Contents

Classes

CategoryTree	Wrapper that maintains flat or hierarchical category information.

class kwcoco.category_tree.**CategoryTree**(*graph=None*)

Bases: ubelt.NiceRepr

Wrapper that maintains flat or hierarchical category information.

Helps compute softmaxes and probabilities for tree-based categories where a directed edge (A, B) represents that A is a superclass of B.

Notes

There are three basic properties that this object maintains:

node: Alphanumeric string names that should be generally descriptive. Using spaces and special characters in these names is discouraged, but can be done. This is the COCO category "name" attribute. For categories this may be denoted as (name, node, cname, catname).

id: The integer id of a category should ideally remain consistent. These are often given by a dataset (e.g. a COCO dataset). This is the COCO category "id" attribute. For categories this is often denoted as (id, cid).

index: Contigous zero-based indices that indexes the list of categories. These should be used for the fastest access in backend computation tasks. Typically corresponds to the ordering of the channels in the final linear layer in an associated model. For categories this is often denoted as (index, cidx, idx, or cx).

Variables

- **idx_to_node** (*List[str]*) – a list of class names. Implicitly maps from index to category name.
- **id_to_node** (*Dict[int, str]*) – maps integer ids to category names
- **node_to_id** (*Dict[str, int]*) – maps category names to ids
- **node_to_idx** (*Dict[str, int]*) – maps category names to indexes
- **graph** (*nx.Graph*) – a Graph that stores any hierarchy information. For standard mutually exclusive classes, this graph is edgeless. Nodes in this graph can maintain category attributes / properties.
- **idx_groups** (*List[List[int]]*) – groups of category indices that share the same parent category.

Example

```
>>> from kwcoco.category_tree import *
>>> graph = nx.from_dict_of_lists({
>>>     'background': [],
>>>     'foreground': ['animal'],
>>>     'animal': ['mammal', 'fish', 'insect', 'reptile'],
>>>     'mammal': ['dog', 'cat', 'human', 'zebra'],
>>>     'zebra': ['grevys', 'plains'],
>>>     'grevys': ['fred'],
>>>     'dog': ['boxer', 'beagle', 'golden'],
>>>     'cat': ['maine coon', 'persian', 'sphynx'],
>>>     'reptile': ['bearded dragon', 't-rex'],
>>> }, nx.DiGraph)
>>> self = CategoryTree(graph)
>>> print(self)
<CategoryTree(nNodes=22, maxDepth=6, maxBreadth=4...)>
```

Example

```
>>> # The coerce classmethod is the easiest way to create an instance
>>> import kwcoco
>>> kwcoco.CategoryTree.coerce(['a', 'b', 'c'])
<CategoryTree(nNodes=3, nodes=['a', 'b', 'c']) ...
>>> kwcoco.CategoryTree.coerce(4)
<CategoryTree(nNodes=4, nodes=['class_1', 'class_2', 'class_3', ...
>>> kwcoco.CategoryTree.coerce(4)
```

copy(*self*)

classmethod from_mutex(*cls*, *nodes*, *bg_hack=True*)

Parameters **nodes** (*List[str]*) – or a list of class names (in which case they will all be assumed to be mutually exclusive)

Example

```
>>> print(CategoryTree.from_mutex(['a', 'b', 'c']))
<CategoryTree(nNodes=3, ...)>
```

classmethod from_json(*cls*, *state*)

Parameters **state** (*Dict*) – see __getstate__ / __json__ for details

classmethod from_coco(*cls*, *categories*)

Create a CategoryTree object from coco categories

Parameters **List[Dict]** – list of coco-style categories

classmethod coerce(*cls*, *data*, ***kw*)

Attempt to coerce data as a CategoryTree object.

This is primarily useful for when the software stack depends on categories being represent

This will work if the input data is a specially formatted json dict, a list of mutually exclusive classes, or if it is already a CategoryTree. Otherwise an error will be thrown.

Parameters

- **data** (*object*) – a known representation of a category tree.
- ****kwargs** – input type specific arguments

Returns self

Return type *CategoryTree*

Raises

- **TypeError - if the input format is unknown** –
- **ValueError - if kwargs are not compatible with the input format** –

Example

```
>>> import kwcoco
>>> classes1 = kwcoco.CategoryTree.coerce(3)  # integer
>>> classes2 = kwcoco.CategoryTree.coerce(classes1.__json__())  # graph dict
>>> classes3 = kwcoco.CategoryTree.coerce(['class_1', 'class_2', 'class_3'])  #␣
↪mutex list
>>> classes4 = kwcoco.CategoryTree.coerce(classes1.graph)  # nx Graph
>>> classes5 = kwcoco.CategoryTree.coerce(classes1)  # cls
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> classes6 = ndsampler.CategoryTree.coerce(3)
>>> classes7 = ndsampler.CategoryTree.coerce(classes1)
>>> classes8 = kwcoco.CategoryTree.coerce(classes6)
```

classmethod demo(*cls*, *key='coco'*, ***kwargs*)

Parameters **key** (*str*) – specify which demo dataset to use. Can be 'coco' (which uses the default coco demo data). Can be 'btree' which creates a binary tree and accepts kwargs

'r' and 'h' for branching-factor and height.

Can be 'btree2', which is the same as btree but returns strings

CommandLine: xdoctest -m ~/code/kwcoco/kwcoco/category_tree.py CategoryTree.demo

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self = {}'.format(self))
self = <CategoryTree(nNodes=10, maxDepth=2, maxBreadth=4...)>
```

to_coco(*self*)

Converts to a coco-style data structure

Yields *Dict* – coco category dictionaries

id_to_idx(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.id_to_idx[1]
```

idx_to_id(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.idx_to_id[0]
```

idx_to_ancestor_idxs(*self*, *include_self=True*)
Mapping from a class index to its ancestors

Parameters **include_self** (*bool, default=True*) – if True includes each node as its own ancestor.

idx_to_descendants_idxs(*self*, *include_self=False*)
Mapping from a class index to its descendants (including itself)

Parameters **include_self** (*bool, default=False*) – if True includes each node as its own descendant.

idx_pairwise_distance(*self*)
Get a matrix encoding the distance from one class to another.

Distances

- from parents to children are positive (descendants),
- from children to parents are negative (ancestors),
- between unreachable nodes (wrt to forward and reverse graph) are

nan.

__len__(*self*)

__iter__(*self*)

__getitem__(*self*, *index*)

__contains__(*self*, *node*)

__json__(*self*)

Example

```
>>> import pickle
>>> self = CategoryTree.demo()
>>> print('self = {!r}'.format(self.__json__()))
```

__getstate__(*self*)

Serializes information in this class

Example

```
>>> from kwcoco.category_tree import *
>>> import pickle
>>> self = CategoryTree.demo()
>>> state = self.__getstate__()
>>> serialization = pickle.dumps(self)
>>> recon = pickle.loads(serialization)
>>> assert recon.__json__() == self.__json__()
```

__setstate__(*self*, *state*)

__nice__(*self*)

is_mutex(*self*)

Returns True if all categories are mutually exclusive (i.e. flat)

If true, then the classes may be represented as a simple list of class names without any loss of information, otherwise the underlying category graph is necessary to preserve all knowledge.

Todo:

- [ ] what happens when we have a dummy root?

---

property num_classes(*self*)

property class_names(*self*)

property category_names(*self*)

property cats(*self*)

Returns a mapping from category names to category attributes.

If this category tree was constructed from a coco-dataset, then this will contain the coco category attributes.

Returns Dict[str, Dict[str, object]]

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self.cats = {!r}'.format(self.cats))
```

index(*self*, *node*)
Return the index that corresponds to the category name

_build_index(*self*)
construct lookup tables

show(*self*)

Ignore:

```
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco import category_tree
>>> self = category_tree.CategoryTree.demo()
>>> self.show()
```

python -c "import kwplot, kwcoco, graphid; kwplot.autompl(); graphid.util.show_nx(kwcoco.category_tree.CategoryTree.demo().graph); kwplot.show_if_requested()" --show

2.3.4 kwcoco.channel_spec

The ChannelSpec has these simple rules:

- each 1D channel is a alphanumeric string.
- The pipe ('|') separates aligned early fused stremas (non-communative)
- The comma (',') separates late-fused streams, (happens after pipe operations, and is communative)
- Certain common sets of early fused channels have codenames, for example:

  rgb = r|g|b rgba = r|g|b|a dxdy = dy|dy

For single arrays, the spec is always an early fused spec.

Todo:

- [X] : normalize representations? e.g: rgb = r|g|b? - OPTIONAL
- [X] : rename to BandsSpec or SensorSpec? - REJECTED
- [ ] : allow bands to be coerced, i.e. rgb -> gray, or gray->rgb

---

Module Contents

Classes

FusedChannelSpec	A specific type of channel spec with only one early fused stream.
ChannelSpec	Parse and extract information about network input channel specs for

Functions

_cached_single_fused_mapping(item_keys, parsed_items, axis=0)	
_cached_single_stream_idxs(key, axis=0)	hack for speed
subsequence_index(oset1, oset2)	Returns a slice into the first items indicating the position of
oset_insert(self, index, obj)	self = ub.oset()
oset_delitem(self, index)	for ubelt oset, todo contribute back to luminosoinsight

class kwcoco.channel_spec.**FusedChannelSpec**(*parsed*)

Bases: ubelt.NiceRepr

A specific type of channel spec with only one early fused stream.

The channels in this stream are non-communative

Notes

This class name and API is in flux and subject to change.

Todo: A special code indicating a name and some number of bands that that names contains, this would primarilly be used for large numbers of channels produced by a network. Like:

resnet_d35d060_L5:512

or

resnet_d35d060_L5[:512]

might refer to a very specific (hashed) set of resnet parameters with 512 bands

maybe we can do something slicly like:

resnet_d35d060_L5[A:B] resnet_d35d060_L5:A:B

Do we want to "just store the code" and allow for parsing later?

Or do we want to ensure the serialization is parsed before we construct the data structure?

_alias_lut

_size_lut

__len__(*self*)

__getitem__(*self*, *index*)

classmethod concat(*cls*, *items*)

spec(*self*)

unique(*self*)

classmethod parse(*cls*, *spec*)

classmethod coerce(*cls*, *data*)

Example

```
>>> FusedChannelSpec.coerce(['a', 'b', 'c'])
>>> FusedChannelSpec.coerce('a|b|c')
>>> FusedChannelSpec.coerce(3)
>>> FusedChannelSpec.coerce(FusedChannelSpec(['a']))
```

__nice__(*self*)

__json__(*self*)

normalize(*self*)
Replace aliases with explicit single-band-per-code specs

Example

```
>>> self = FusedChannelSpec.coerce('b1|b2|b3|rgb')
>>> normed = self.normalize()
>>> print('normed = {}'.format(ub.repr2(normed, nl=1)))
```

__contains__(*self*, *key*)

Example

```
>>> FCS = FusedChannelSpec.coerce
>>> 'disparity' in FCS('rgb|disparity|flowx|flowy')
True
>>> 'gray' in FCS('rgb|disparity|flowx|flowy')
False
```

code_list(*self*)
Return the expanded code list

as_list(*self*)

as_oset(*self*)

as_set(*self*)

difference(*self*, *other*)
Set difference

Example

```
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('rgb|disparity|flowx|flowy')
>>> other = FCS('r|b')
>>> self.difference(other)
>>> other = FCS('flowx')
>>> self.difference(other)
```

intersection(*self*, *other*)

Example

```
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('rgb|disparity|flowx|flowy')
>>> other = FCS('r|b|XX')
>>> self.intersection(other)
```

component_indices(*self*, *axis=2*)

Look up component indices within this stream

Example

```
>>> FCS = FusedChannelSpec.coerce
>>> self = FCS('disparity|rgb|flowx|flowy')
>>> component_indices = self.component_indices()
>>> print('component_indices = {}'.format(ub.repr2(component_indices, nl=1)))
component_indices = {
    'disparity': (slice(...), slice(...), slice(0, 1, None)),
    'flowx': (slice(...), slice(...), slice(4, 5, None)),
    'flowy': (slice(...), slice(...), slice(5, 6, None)),
    'rgb': (slice(...), slice(...), slice(1, 4, None)),
}
```

class kwcoco.channel_spec.**ChannelSpec**(*spec*, *parsed=None*)

Bases: ubelt.NiceRepr

Parse and extract information about network input channel specs for early or late fusion networks.

Notes

This class name and API is in flux and subject to change.

Notes

The pipe ('|') character represents an early-fused input stream, and order matters (it is non-communative).

The comma (',') character separates different inputs streams/branches for a multi-stream/branch network which will be lated fused. Order does not matter

Example

```
>>> # Integer spec
>>> ChannelSpec.coerce(3)
<ChannelSpec(u0|u1|u2) ...>
```

```
>>> # single mode spec
>>> ChannelSpec.coerce('rgb')
<ChannelSpec(rgb) ...>
```

```
>>> # early fused input spec
>>> ChannelSpec.coerce('rgb|disprity')
<ChannelSpec(rgb|disprity) ...>
```

```
>>> # late fused input spec
>>> ChannelSpec.coerce('rgb,disprity')
<ChannelSpec(rgb,disprity) ...>
```

```
>>> # early and late fused input spec
>>> ChannelSpec.coerce('rgb|ir,disprity')
<ChannelSpec(rgb|ir,disprity) ...>
```

Example

```
>>> self = ChannelSpec('gray')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb|disparity')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb|disparity,disparity')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>> self = ChannelSpec('rgb,disparity,flowx|flowy')
>>> print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
```

Example

```
>>> specs = [
>>>     'rgb',              # and rgb input
>>>     'rgb|disprity',     # rgb early fused with disparity
>>>     'rgb,disprity',     # rgb early late with disparity
>>>     'rgb|ir,disprity',  # rgb early fused with ir and late fused with disparity
>>>     3,                  # 3 unknown channels
>>> ]
>>> for spec in specs:
>>>     print('======================')
>>>     print('spec = {!r}'.format(spec))
>>>     #
>>>     self = ChannelSpec.coerce(spec)
>>>     print('self = {!r}'.format(self))
>>>     sizes = self.sizes()
>>>     print('sizes = {!r}'.format(sizes))
>>>     print('self.info = {}'.format(ub.repr2(self.info, nl=1)))
>>>     #
>>>     item = self._demo_item((1, 1), rng=0)
>>>     inputs = self.encode(item)
>>>     components = self.decode(inputs)
>>>     input_shapes = ub.map_vals(lambda x: x.shape, inputs)
>>>     component_shapes = ub.map_vals(lambda x: x.shape, components)
>>>     print('item = {}'.format(ub.repr2(item, precision=1)))
>>>     print('inputs = {}'.format(ub.repr2(inputs, precision=1)))
>>>     print('input_shapes = {}'.format(ub.repr2(input_shapes)))
>>>     print('components = {}'.format(ub.repr2(components, precision=1)))
>>>     print('component_shapes = {}'.format(ub.repr2(component_shapes, nl=1)))
```

_alias_lut

_size_lut

__nice__(*self*)

__json__(*self*)

__contains__(*self*, *key*)

Example

```
>>> 'disparity' in ChannelSpec('rgb,disparity,flowx|flowy')
True
>>> 'gray' in ChannelSpec('rgb,disparity,flowx|flowy')
False
```

property info(*self*)

classmethod coerce(*cls*, *data*)

parse(*self*)
Build internal representation

normalize(*self*)
Replace aliases with explicit single-band-per-code specs

Example

```
>>> self = ChannelSpec('b1|b2|b3|rgb')
>>> self.normalize()
>>> list(self.keys())
```

keys(*self*)

values(*self*)

items(*self*)

streams(*self*)
Breaks this spec up into one spec for each early-fused input stream

code_list(*self*)

difference(*self*, *other*)
Set difference

Example

```
>>> self = ChannelSpec('rgb|disparity,flowx|flowy')
>>> other = ChannelSpec('rgb')
>>> self.difference(other)
>>> other = ChannelSpec('flowx')
>>> self.difference(other)
```

sizes(*self*)
Number of dimensions for each fused stream channel

IE: The EARLY-FUSED channel sizes

Example

```
>>> self = ChannelSpec('rgb|disparity,flowx|flowy')
>>> self.sizes()
```

unique(*self*, *normalize=False*)
Returns the unique channels that will need to be given or loaded

_item_shapes(*self*, *dims*)
Expected shape for an input item

Parameters **dims** (*Tuple[int, int]*) – the spatial dimension

Returns Dict[int, tuple]

_demo_item(*self*, *dims=(4, 4)*, *rng=None*)
Create an input that satisfies this spec

Returns

an item like it might appear when its returned from the *__getitem__* method of a `torch...Dataset`.

Return type dict

Example

```
>>> dims = (1, 1)
>>> ChannelSpec.coerce(3)._demo_item(dims, rng=0)
>>> ChannelSpec.coerce('r|g|b|disaprity')._demo_item(dims, rng=0)
>>> ChannelSpec.coerce('rgb|disaprity')._demo_item(dims, rng=0)
>>> ChannelSpec.coerce('rgb,disaprity')._demo_item(dims, rng=0)
>>> ChannelSpec.coerce('rgb')._demo_item(dims, rng=0)
>>> ChannelSpec.coerce('gray')._demo_item(dims, rng=0)
```

encode(*self*, *item*, *axis=0*, *mode=1*)

Given a dictionary containing preloaded components of the network inputs, build a concatenated (fused) network representations of each input stream.

Parameters

- **item** (*Dict[str, Tensor]*) – a batch item containing unfused parts. each key should be a single-stream (optionally early fused) channel key.
- **axis** (*int, default=0*) – concatenation dimension

Returns mapping between input stream and its early fused tensor input.

Return type Dict[str, Tensor]

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> import numpy as np
>>> dims = (4, 4)
>>> item = {
>>>     'rgb': np.random.rand(3, *dims),
>>>     'disparity': np.random.rand(1, *dims),
>>>     'flowx': np.random.rand(1, *dims),
>>>     'flowy': np.random.rand(1, *dims),
>>> }
>>> # Complex Case
>>> self = ChannelSpec('rgb,disparity,rgb|disparity|flowx|flowy,flowx|flowy')
>>> fused = self.encode(item)
>>> input_shapes = ub.map_vals(lambda x: x.shape, fused)
>>> print('input_shapes = {}'.format(ub.repr2(input_shapes, nl=1)))
>>> # Simpler case
>>> self = ChannelSpec('rgb|disparity')
>>> fused = self.encode(item)
>>> input_shapes = ub.map_vals(lambda x: x.shape, fused)
>>> print('input_shapes = {}'.format(ub.repr2(input_shapes, nl=1)))
```

Example

```
>>> # Case where we have to break up early fused data
>>> import numpy as np
>>> dims = (40, 40)
>>> item = {
>>>     'rgb|disparity': np.random.rand(4, *dims),
>>>     'flowx': np.random.rand(1, *dims),
>>>     'flowy': np.random.rand(1, *dims),
>>> }
>>> # Complex Case
>>> self = ChannelSpec('rgb,disparity,rgb|disparity,rgb|disparity|flowx|flowy,
↪flowx|flowy,flowx,disparity')
>>> inputs = self.encode(item)
>>> input_shapes = ub.map_vals(lambda x: x.shape, inputs)
>>> print('input_shapes = {}'.format(ub.repr2(input_shapes, nl=1)))
```

```
>>> # xdoctest: +REQUIRES(--bench)
>>> #self = ChannelSpec('rgb|disparity,flowx|flowy')
>>> import timerit
>>> ti = timerit.Timerit(100, bestof=10, verbose=2)
>>> for timer in ti.reset('mode=simple'):
>>>     with timer:
>>>         inputs = self.encode(item, mode=0)
>>> for timer in ti.reset('mode=minimize-concat'):
>>>     with timer:
>>>         inputs = self.encode(item, mode=1)
```

import xdev _ = xdev.profile_now(self.encode)(item, mode=1)

decode(*self*, *inputs*, *axis=1*)

break an early fused item into its components

Parameters

- **inputs** (*Dict[str, Tensor]*) – dictionary of components
- **axis** (*int, default=1*) – channel dimension

Example

```
>>> from kwcoco.channel_spec import *  # NOQA
>>> import numpy as np
>>> dims = (4, 4)
>>> item_components = {
>>>     'rgb': np.random.rand(3, *dims),
>>>     'ir': np.random.rand(1, *dims),
>>> }
>>> self = ChannelSpec('rgb|ir')
>>> item_encoded = self.encode(item_components)
>>> batch = {k: np.concatenate([v[None, :], v[None, :]], axis=0)
...          for k, v in item_encoded.items()}
>>> components = self.decode(batch)
```

Example

```
>>> # xdoctest: +REQUIRES(module:netharn, module:torch)
>>> import torch
>>> import numpy as np
>>> dims = (4, 4)
>>> components = {
>>>     'rgb': np.random.rand(3, *dims),
>>>     'ir': np.random.rand(1, *dims),
>>> }
>>> components = ub.map_vals(torch.from_numpy, components)
>>> self = ChannelSpec('rgb|ir')
>>> encoded = self.encode(components)
>>> from netharn.data import data_containers
>>> item = {k: data_containers.ItemContainer(v, stack=True)
>>>         for k, v in encoded.items()}
>>> batch = data_containers.container_collate([item, item])
>>> components = self.decode(batch)
```

component_indices(*self*, *axis=2*)

Look up component indices within fused streams

Example

```
>>> dims = (4, 4)
>>> inputs = ['flowx', 'flowy', 'disparity']
>>> self = ChannelSpec('disparity,flowx|flowy')
>>> component_indices = self.component_indices()
>>> print('component_indices = {}'.format(ub.repr2(component_indices, nl=1)))
component_indices = {
    'disparity': ('disparity', (slice(None, None, None), slice(None, None,␣
↪None), slice(0, 1, None))),
    'flowx': ('flowx|flowy', (slice(None, None, None), slice(None, None, None),␣
↪slice(0, 1, None))),
    'flowy': ('flowx|flowy', (slice(None, None, None), slice(None, None, None),␣
↪slice(1, 2, None))),
}
```

kwcoco.channel_spec.**_cached_single_fused_mapping**(*item_keys*, *parsed_items*, *axis=0*)

kwcoco.channel_spec.**_cached_single_stream_idxs**(*key*, *axis=0*)

hack for speed

axis = 0 key = 'rgb|disparity'

xdoctest: +REQUIRES(–bench) import timerit ti = timerit.Timerit(100, bestof=10, verbose=2) for timer in ti.reset('time'):

with timer: _cached_single_stream_idxs(key, axis=axis)

for timer in ti.reset('time'):

with timer: ChannelSpec(key).component_indices(axis=axis)

kwcoco.channel_spec.**subsequence_index**(*oset1*, *oset2*)
Returns a slice into the first items indicating the position of the second items if they exist.

This is a variant of the substring problem.

Returns None | slice

Example

```
>>> oset1 = ub.oset([1, 2, 3, 4, 5, 6])
>>> oset2 = ub.oset([2, 3, 4])
>>> index = subsequence_index(oset1, oset2)
>>> assert index
```

```
>>> oset1 = ub.oset([1, 2, 3, 4, 5, 6])
>>> oset2 = ub.oset([2, 4, 3])
>>> index = subsequence_index(oset1, oset2)
>>> assert not index
```

kwcoco.channel_spec.**oset_insert**(*self*, *index*, *obj*)
self = ub.oset() oset_insert(self, 0, 'a') oset_insert(self, 0, 'b') oset_insert(self, 0, 'c') oset_insert(self, 1, 'd') oset_insert(self, 2, 'e') oset_insert(self, 0, 'f')

kwcoco.channel_spec.**oset_delitem**(*self*, *index*)
for ubelt oset, todo contribute back to luminosoinsight

```
>>> self = ub.oset([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> index = slice(3, 5)
>>> oset_delitem(self, index)
```

self = ub.oset(['r', 'g', 'b', 'disparity']) index = slice(0, 3) oset_delitem(self, index)

2.3.5 kwcoco.coco_dataset

An implementation and extension of the original MS-COCO API[1].

Extends the format to also include line annotations.

The following describes psuedo-code for the high level spec (some of which may not be have full support in the Python API). A formal json-schema is defined in **:module:`kwcoco.coco_schema.py`**.

An informal spec is as follows:

```
# All object categories are defined here.
category = {
    'id': int,
    'name': str,  # unique name of the category
    'supercategory': str,   # parent category name
}

# Videos are used to manage collections or sequences of images.
# Frames do not necesarilly have to be aligned or uniform time steps
video = {
```

(continues on next page)

[1] http://cocodataset.org/#format-data

(continued from previous page)

```
    'id': int,
    'name': str,  # a unique name for this video.

    'width': int  # the base width of this video (all associated images must have this␣
↪width)
    'height': int  # the base height of this video (all associated images must have this␣
↪height)

    # In the future this may be extended to allow pointing to video files
}

# Specifies how to find sensor data of a particular scene at a particular
# time. This is usually paths to rgb images, but auxiliary information
# can be used to specify multiple bands / etc...
image = {
    'id': int,

    'name': str,  # an encouraged but optional unique name
    'file_name': str,  # relative path to the "base" image data

    'width': int,   # pixel width of "base" image
    'height': int,  # pixel height of "base" image

    'channels': <ChannelSpec>,   # a string encoding of the channels in the main image

    'auxiliary': [  # information about any auxiliary channels / bands
        {
            'file_name': str,     # relative path to associated file
            'channels': <ChannelSpec>,   # a string encoding
            'width':     <int>    # pixel width of auxiliary image
            'height':    <int>    # pixel height of auxiliary image
            'warp_aux_to_img': <TransformSpec>,  # tranform from "base" image space to␣
↪auxiliary image space. (identity if unspecified)
        }, ...
    ]

    'video_id': str  # if this image is a frame in a video sequence, this id is shared␣
↪by all frames in that sequence.
    'timestamp': str | int  # a iso-string timestamp or an integer in flicks.
    'frame_index': int  # ordinal frame index which can be used if timestamp is unknown.
    'warp_img_to_vid': <TransformSpec>  # a transform image space to video space␣
↪(identity if unspecified), can be used for sensor alignment or video stabilization
}

TransformSpec:
    The spec can be anything coercable to a kwimage.Affine object.
    This can be an explicit affine transform matrix like:
        {'type': 'affine': 'matrix': <a-3x3 matrix>},

    But it can also be a concise dict containing one or more of these keys
        {
            'scale': <float|Tuple[float, float]>,
```

(continues on next page)

(continued from previous page)

```
            'offset': <float|Tuple[float, float]>,
            'skew': <float>,
            'theta': <float>,  # radians counter-clock-wise
        }

ChannelSpec:
    This is a string that describes the channel composition of an image.
    For the purposes of kwcoco, separate different channel names with a
    pipe ('|'). If the spec is not specified, methods may fall back on
    grayscale or rgb processing. There are special string. For instance
    'rgb' will expand into 'r|g|b'. In other applications you can "late
    fuse" inputs by separating them with a "," and "early fuse" by
    separating with a "|". Early fusion returns a solid array/tensor, late
    fusion returns separated arrays/tensors.

# Ground truth is specified as annotations, each belongs to a spatial
# region in an image. This must reference a subregion of the image in pixel
# coordinates. Additional non-schma properties can be specified to track
# location in other coordinate systems. Annotations can be linked over time
# by specifying track-ids.
annotation = {
    'id': int,
    'image_id': int,
    'category_id': int,

    'track_id': <int | str | uuid>  # indicates association between annotations across␣
↪images

    'bbox': [tl_x, tl_y, w, h],  # xywh format)
    'score' : float,
    'prob' : List[float],
    'weight' : float,

    'caption': str,  # a text caption for this annotation
    'keypoints' : <Keypoints | List[int] > # an accepted keypoint format
    'segmentation': <RunLengthEncoding | Polygon | MaskPath | WKT >,  # an accepted␣
↪segmentation format
}

# A dataset bundles a manifest of all aformentioned data into one structure.
dataset = {
    'categories': [category, ...],
    'videos': [video, ...]
    'images': [image, ...]
    'annotations': [annotation, ...]
    'licenses': [],
    'info': [],
}

Polygon:
    A flattned list of xy coordinates.
    [x1, y1, x2, y2, ..., xn, yn]
```

(continues on next page)

(continued from previous page)

```
    or a list of flattned list of xy coordinates if the CCs are disjoint
    [[x1, y1, x2, y2, ..., xn, yn], [x1, y1, ..., xm, ym],]

    Note: the original coco spec does not allow for holes in polygons.

    We also allow a non-standard dictionary encoding of polygons
        {'exterior': [(x1, y1)...],
         'interiors': [[(x1, y1), ...], ...]}

    TODO: Support WTK

RunLengthEncoding:
    The RLE can be in a special bytes encoding or in a binary array
    encoding. We reuse the original C functions are in [2]_ in
    ``kwimage.structs.Mask`` to provide a convinient way to abstract this
    rather esoteric bytes encoding.

    For pure python implementations see kwimage:
        Converting from an image to RLE can be done via kwimage.run_length_encoding
        Converting from RLE back to an image can be done via:
            kwimage.decode_run_length

        For compatibility with the COCO specs ensure the binary flags
        for these functions are set to true.

Keypoints:
    Annotation keypoints may also be specified in this non-standard (but
    ultimately more general) way:

    'annotations': [
        {
            'keypoints': [
                {
                    'xy': <x1, y1>,
                    'visible': <0 or 1 or 2>,
                    'keypoint_category_id': <kp_cid>,
                    'keypoint_category': <kp_name, optional>,  # this can be specified␣
↪instead of an id
                }, ...
            ]
        }, ...
    ],
    'keypoint_categories': [{
        'name': <str>,
        'id': <int>,  # an id for this keypoint category
        'supercategory': <kp_name>  # name of coarser parent keypoint class (for␣
↪hierarchical keypoints)
        'reflection_id': <kp_cid>  # specify only if the keypoint id would be swapped␣
↪with another keypoint type
    },...
    ]
```

(continues on next page)

(continued from previous page)

```
    In this scheme the "keypoints" property of each annotation (which used
    to be a list of floats) is now specified as a list of dictionaries that
    specify each keypoints location, id, and visibility explicitly. This
    allows for things like non-unique keypoints, partial keypoint
    annotations. This also removes the ordering requirement, which makes it
    simpler to keep track of each keypoints class type.

    We also have a new top-level dictionary to specify all the possible
    keypoint categories.

    TODO: Support WTK

Auxiliary Channels:
    For multimodal or multispectral images it is possible to specify
    auxiliary channels in an image dictionary as follows:

    {
        'id': int,
        'file_name': str,    # path to the "base" image (may be None)
        'name': str,         # a unique name for the image (must be given if file_name␣
↪is None)
        'channels': <spec>,  # a spec code that indicates the layout of the "base" image␣
↪channels.
        'auxiliary': [  # information about auxiliary channels
            {
                'file_name': str,
                'channels': <spec>
            }, ... # can have many auxiliary channels with unique specs
        ]
    }

Video Sequences:
    For video sequences, we add the following video level index:

    'videos': [
        { 'id': <int>, 'name': <video_name:str> },
    ]

    Note that the videos might be given as encoded mp4/avi/etc.. files (in
    which case the name should correspond to a path) or as a series of
    frames in which case the images should be used to index the extracted
    frames and information in them.

    Then image dictionaries are augmented as follows:

    {
        'video_id': str  # optional, if this image is a frame in a video sequence, this␣
↪id is shared by all frames in that sequence.
        'timestamp': int  # optional, timestamp (ideally in flicks), used to identify␣
↪the timestamp of the frame. Only applicable video inputs.
        'frame_index': int  # optional, ordinal frame index which can be used if␣
↪timestamp is unknown.
```

(continues on next page)

(continued from previous page)

```
    }

    And annotations are augmented as follows:

    {
        'track_id': <int | str | uuid>  # optional, indicates association between␣
↪annotations across frames
    }
```

Notes

The main object in this file is class:*CocoDataset*, which is composed of several mixin classes. See the class and method documentation for more details.

Todo:

- [ ] Use ijson to lazilly load pieces of the dataset in the background or on demand. This will give us faster access to categories / images, whereas we will always have to wait for annotations etc…
- [X] Should img_root be changed to bundle_dpath?
- [ ] Read video data, return numpy arrays (requires API for images)
- [ ] Spec for video URI, and convert to frames @ framerate function.
- [ ] Document channel spec
- [X] remove videos

References

Module Contents

Classes

MixinCocoDepricate	These functions are marked for deprication and may be removed at any time
MixinCocoAccessors	TODO: better name
MixinCocoExtras	Misc functions for coco
MixinCocoObjects	Expose methods to construct object lists / groups.
MixinCocoStats	Methods for getting stats about the dataset
_NextId	Helper class to tracks unused ids for new items
_ID_Remapper	Helper to recycle ids for unions.
UniqueNameRemapper	helper to ensure names will be unique by appending suffixes
MixinCocoDraw	Matplotlib / display functionality
MixinCocoAddRemove	Mixin functions to dynamically add / remove annotations images and
CocoIndex	Fast lookup index for the COCO dataset with dynamic modification

continues on next page

Table 77 – continued from previous page

MixinCocoIndex	Give the dataset top level access to index attributes
CocoDataset	

Notes

Functions

_delitems(items, remove_idxs, thresh=750)	**Parameters** • **items** (*List*) – list which will be modified
demo_coco_data()	Simple data for testing.

Attributes

_dict
SPEC_KEYS
INT_TYPES

kwcoco.coco_dataset.**_dict**

kwcoco.coco_dataset.**SPEC_KEYS = ['info', 'licenses', 'categories', 'keypoint_categories', 'videos', 'images', 'annotations']**

kwcoco.coco_dataset.**INT_TYPES**

class kwcoco.coco_dataset.**MixinCocoDepricate**

Bases: object

These functions are marked for deprication and may be removed at any time

class kwcoco.coco_dataset.**MixinCocoAccessors**

Bases: object

TODO: better name

delayed_load(*self*, *gid*, *channels=None*, *space='image'*)

Experimental method

Parameters

- **gid** (*int*) – image id to load
- **channels** (*FusedChannelSpec*) – specific channels to load. if unspecified, all channels are loaded.
- **space** (*str*) – can either be “image” for loading in image space, or “video” for loading in video space.

Todo:

- **[ ] Currently can only take all or none of the channels from each** base-image / auxiliary dict. For instance if the main image is r|g|b you can't just select g|b at the moment.
- **[ ] The order of the channels in the delayed load should** match the requested channel order.
- [ ] TODO: add nans to bands that don't exist or throw an error

Example

```
>>> import kwcoco
>>> gid = 1
>>> #
>>> self = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = self.delayed_load(gid)
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize(as_xarray=True)))
>>> #
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> delayed = self.delayed_load(gid)
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize()))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize(as_xarray=True)))
```

```
>>> crop = delayed.delayed_crop((slice(0, 3), slice(0, 3)))
>>> crop.finalize()
>>> crop.finalize(as_xarray=True)
```

```
>>> # TODO: should only select the "red" channel
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> delayed = self.delayed_load(gid, channels='r')
```

```
>>> import kwcoco
>>> gid = 1
>>> #
>>> self = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> delayed = self.delayed_load(gid, channels='B1|B2', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize(as_xarray=True)))
>>> delayed = self.delayed_load(gid, channels='B1|B2|B11', space='image')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize(as_xarray=True)))
>>> delayed = self.delayed_load(gid, channels='B8|B1', space='video')
>>> print('delayed = {!r}'.format(delayed))
>>> print('delayed.finalize() = {!r}'.format(delayed.finalize(as_xarray=True)))
```

load_image(*self*, *gid_or_img*, *channels=None*)

Reads an image from disk and

Parameters

- **gid_or_img** (*int or dict*) – image id or image dict
- **channels** (*str | None*) – if specified, load data from auxiliary channels instead

Returns the image

Return type np.ndarray

Todo:

- [ ] allow specification of multiple channels

---

get_image_fpath(*self*, *gid_or_img*, *channels=None*)
Returns the full path to the image

Parameters

- **gid_or_img** (*int or dict*) – image id or image dict
- **channels** (*str, default=None*) – if specified, return a path to data containing auxiliary channels instead

Returns full path to the image

Return type PathLike

_get_img_auxiliary(*self*, *gid_or_img*, *channels*)
returns the auxiliary dictionary for a specific channel

get_auxiliary_fpath(*self*, *gid_or_img*, *channels*)
Returns the full path to auxiliary data for an image

Parameters

- **gid_or_img** (*int | dict*) – an image or its id
- **channels** (*str*) – the auxiliary channel to load (e.g. disparity)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8', aux=True)
>>> self.get_auxiliary_fpath(1, 'disparity')
```

load_annot_sample(*self*, *aid_or_ann*, *image=None*, *pad=None*)
Reads the chip of an annotation. Note this is much less efficient than using a sampler, but it doesn't require disk cache.

Parameters

- **aid_or_int** (*int or dict*) – annot id or dict
- **image** (*ArrayLike, default=None*) – preloaded image (note: this process is inefficient unless image is specified)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> sample = self.load_annot_sample(2, pad=100)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'])
>>> kwplot.show_if_requested()
```

_resolve_to_id(*self*, *id_or_dict*)
Ensures output is an id

_resolve_to_cid(*self*, *id_or_name_or_dict*)
Ensures output is an category id

Note: this does not resolve aliases (yet), for that see _alias_to_cat Todo: we could maintain an alias index to make this fast

_resolve_to_gid(*self*, *id_or_name_or_dict*)
Ensures output is an category id

_resolve_to_vidid(*self*, *id_or_name_or_dict*)
Ensures output is an video id

_resolve_to_ann(*self*, *aid_or_ann*)
Ensures output is an annotation dictionary

_resolve_to_img(*self*, *gid_or_img*)
Ensures output is an image dictionary

_resolve_to_kpcat(*self*, *kp_identifier*)
Lookup a keypoint-category dict via its name or id

Parameters **kp_identifier** (*int | str | dict*) – either the keypoint category name, alias, or its keypoint_category_id.

Returns keypoint category dictionary

Return type Dict

Example

```
>>> self = CocoDataset.demo('shapes')
>>> kpcat1 = self._resolve_to_kpcat(1)
>>> kpcat2 = self._resolve_to_kpcat('left_eye')
>>> assert kpcat1 is kpcat2
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     self._resolve_to_cat('human')
```

_resolve_to_cat(*self*, *cat_identifier*)
Lookup a coco-category dict via its name, alias, or id.

Parameters **cat_identifier** (*int | str | dict*) – either the category name, alias, or its category_id.

Raises **KeyError** – if the category doesn't exist.

Notes

If the index is not built, the method will work but may be slow.

Example

```
>>> self = CocoDataset.demo()
>>> cat = self._resolve_to_cat('human')
>>> import pytest
>>> assert self._resolve_to_cat(cat['id']) is cat
>>> assert self._resolve_to_cat(cat) is cat
>>> with pytest.raises(KeyError):
>>>     self._resolve_to_cat(32)
>>> self.index.clear()
>>> assert self._resolve_to_cat(cat['id']) is cat
>>> with pytest.raises(KeyError):
>>>     self._resolve_to_cat(32)
```

_alias_to_cat(*self*, *alias_catname*)

Lookup a coco-category via its name or an "alias" name. In production code, use **:method:`_resolve_to_cat`** instead.

Parameters **alias_catname** (*str*) – category name or alias

Returns coco category dictionary

Return type dict

Example

```
>>> self = CocoDataset.demo()
>>> cat = self._alias_to_cat('human')
>>> import pytest
>>> with pytest.raises(KeyError):
>>>     self._alias_to_cat('person')
>>> cat['alias'] = ['person']
>>> self._alias_to_cat('person')
>>> cat['alias'] = 'person'
>>> self._alias_to_cat('person')
>>> assert self._alias_to_cat(None) is None
```

category_graph(*self*)

Construct a networkx category hierarchy

Returns

graph: a directed graph where category names are the nodes, supercategories define edges, and items in each category dict (e.g. category id) are added as node properties.

Return type network.DiGraph

Example

```
>>> self = CocoDataset.demo()
>>> graph = self.category_graph()
>>> assert 'astronaut' in graph.nodes()
>>> assert 'keypoints' in graph.nodes['human']
```

import graphid graphid.util.show_nx(graph)

object_categories(*self*)

Construct a consistent CategoryTree representation of object classes

Returns category data structure

Return type *kwcoco.CategoryTree*

Example

```
>>> self = CocoDataset.demo()
>>> classes = self.object_categories()
>>> print('classes = {}'.format(classes))
```

keypoint_categories(*self*)

Construct a consistent CategoryTree representation of keypoint classes

Returns category data structure

Return type *kwcoco.CategoryTree*

Example

```
>>> self = CocoDataset.demo()
>>> classes = self.keypoint_categories()
>>> print('classes = {}'.format(classes))
```

_keypoint_category_names(*self*)

Construct keypoint categories names.

Uses new-style if possible, otherwise this falls back on old-style.

Returns names: list of keypoint category names

Return type List[str]

Example

```
>>> self = CocoDataset.demo()
>>> names = self._keypoint_category_names()
>>> print(names)
```

_lookup_kpnames(*self*, *cid*)

Get the keypoint categories for a certain class

_coco_image(*self*, *gid*)

class kwcoco.coco_dataset.**MixinCocoExtras**

Bases: object

Misc functions for coco

classmethod coerce(*cls*, *key*, ***kw*)

Attempt to transform the input into the intended CocoDataset.

Parameters

- **key** – this can either be an instance of a CocoDataset, a string URI pointing to an on-disk dataset, or a special key for creating demodata.
- ****kw** – passed to whatever constructor is chosen (if any)

Example

```
>>> # test coerce for various input methods
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import assert_dsets_allclose
>>> dct_dset = kwcoco.CocoDataset.coerce('special:shapes8')
>>> copy1 = kwcoco.CocoDataset.coerce(dct_dset)
>>> copy2 = kwcoco.CocoDataset.coerce(dct_dset.fpath)
>>> assert assert_dsets_allclose(dct_dset, copy1)
>>> assert assert_dsets_allclose(dct_dset, copy2)
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> sql_dset = dct_dset.view_sql()
>>> copy3 = kwcoco.CocoDataset.coerce(sql_dset)
>>> copy4 = kwcoco.CocoDataset.coerce(sql_dset.fpath)
>>> assert assert_dsets_allclose(dct_dset, sql_dset)
>>> assert assert_dsets_allclose(dct_dset, copy3)
>>> assert assert_dsets_allclose(dct_dset, copy4)
```

classmethod demo(*cls*, *key='photos'*, ***kw*)

Create a toy coco dataset for testing and demo puposes

Parameters

- **key** (*str*) – either ‘photos’, ‘shapes’, or ‘vidshapes’. There are also undocumented sufixes that can control behavior. TODO: better documentation for these demo datasets.
- ****kw** – if key is shapes, these arguments are passed to toydata generation. The Kwargs section of this docstring documents a subset of the available options. For full details, see `demodata_toy_dset()` and `random_video_dset()`.

Kwargs: image_size (Tuple[int, int]): width / height size of the images

dpath (str): path to the output image directory, defaults to using kwcoco cache dir.

aux (bool): if True generates dummy auxiliary channels

rng (int | RandomState, default=0): random number generator or seed

verbose (int, default=3): verbosity mode

Example

```
>>> print(CocoDataset.demo('photos'))
>>> print(CocoDataset.demo('shapes', verbose=1))
>>> print(CocoDataset.demo('vidshapes', verbose=1))
```

```
>>> print(CocoDataset.demo('shapes256', verbose=0))
>>> print(CocoDataset.demo('shapes8', verbose=0))
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes5', num_frames=5,
>>>                                verbose=0, rng=None)
>>> dset = kwcoco.CocoDataset.demo('vidshapes5', num_frames=5,
>>>                                num_tracks=4, verbose=0, rng=44)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> pnums = kwplot.PlotNums(nSubplots=len(dset.index.imgs))
>>> fnum = 1
>>> for gx, gid in enumerate(dset.index.imgs.keys()):
>>>     canvas = dset.draw_image(gid=gid)
>>>     kwplot.imshow(canvas, pnum=pnums[gx], fnum=fnum)
>>>     #dset.show_image(gid=gid, pnum=pnums[gx])
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes5-aux', num_frames=1,
>>>                                verbose=0, rng=None)
```

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes1-multispectral', num_frames=5,
>>>                                verbose=0, rng=None)
>>> # This is the first use-case of image names
>>> assert len(dset.index.file_name_to_img) == 0, (
>>>     'the multispectral demo case has no "base" image')
>>> assert len(dset.index.name_to_img) == len(dset.index.imgs) == 5
>>> dset.remove_images([1])
>>> assert len(dset.index.name_to_img) == len(dset.index.imgs) == 4
>>> dset.remove_videos([1])
>>> assert len(dset.index.name_to_img) == len(dset.index.imgs) == 0
```

Ignore:

```
>>> # Test the cache
>>> key = 'vidshapes8-aux'
>>> kw = {}
>>> self1 = cls.demo(key=key, use_cache=False, **kw)
>>> self2 = cls.demo(key=key, use_cache=True, **kw)
>>> self2 = cls.demo(key=key, use_cache=True, **kw)
>>> # The non-cached version should be exactly the same as the
>>> # cached version.
>>> assert self1.fpath == self2.fpath
>>> assert self1.dataset == self2.dataset
>>> assert self1.dataset is not self2.dataset
```

Ignore: import xdev key = 'shapes' globals().update(xdev.get_func_kwargs(kwcoco.CocoDataset.demo)) kw = {}

kwcoco.CocoDataset.demo('vidshapes')

_tree(*self*)

developer helper

Ignore: import kwcoco self = kwcoco.CocoDataset.demo('photos') print('self = {!r}'.format(self)) self._tree()

self = kwcoco.CocoDataset.demo('shapes1') print(self.imgs[1]) print('self = {!r}'.format(self)) self._tree()

self = kwcoco.CocoDataset.demo('vidshapes2') print('self = {!r}'.format(self)) print(self.imgs[1]) self._tree()

classmethod random(*cls*, *rng=None*)

Creates a random CocoDataset according to distribution parameters

_build_hashid(*self*, *hash_pixels=False*, *verbose=0*)

Construct a hash that uniquely identifies the state of this dataset.

Parameters

- **hash_pixels** (*bool, default=False*) – If False the image data is not included in the hash, which can speed up computation, but is not 100% robust.
- **verbose** (*int*) – verbosity level

Example

```
>>> self = CocoDataset.demo()
>>> self._build_hashid(hash_pixels=True, verbose=3)
...
>>> # Note: kwimage has changes the name of carl.png to carl.jpg
>>> # in 0.7.0, so that modifies some of the hash. Once 0.7.0
>>> # is landed, we can update this test to re-check for
>>> # those hashes.
>>> print('self.hashid_parts = ' + ub.repr2(self.hashid_parts))
>>> print('self.hashid = {!r}'.format(self.hashid))
self.hashid_parts = {
    'annotations': {
        'json': 'e573f49da7b76e27d0...',
```

(continues on next page)

(continued from previous page)

```
        'num': 11,
    },
    'images': {
        'pixels': '67d741fefc8...',
        'json': '...',
        'num': 3,
    },
    'categories': {
        'json': '82d22e0079...',
        'num': 8,
    },
}
self.hashid = '...
```

Old 'json': '6a446126490aa...', self.hashid = '4769119614e921...

New json': '2221c71496a0... self.hashid = '77d445f05...

Doctest:

```
>>> self = CocoDataset.demo()
>>> self._build_hashid(hash_pixels=True, verbose=3)
>>> self.hashid_parts
>>> # Test that when we modify the dataset only the relevant
>>> # hashid parts are recomputed.
>>> orig = self.hashid_parts['categories']['json']
>>> self.add_category('foobar')
>>> assert 'categories' not in self.hashid_parts
>>> self.hashid_parts
>>> self.hashid_parts['images']['json'] = 'should not change'
>>> self._build_hashid(hash_pixels=True, verbose=3)
>>> assert self.hashid_parts['categories']['json']
>>> assert self.hashid_parts['categories']['json'] != orig
>>> assert self.hashid_parts['images']['json'] == 'should not change'
```

_invalidate_hashid(*self*, *parts=None*)

Called whenever the coco dataset is modified. It is possible to specify which parts were modified so unmodified parts can be reused next time the hash is constructed.

_ensure_imgsize(*self*, *workers=0*, *verbose=1*, *fail=False*)

Populate the imgsize field if it does not exist.

Parameters

- **workers** (*int, default=0*) – number of workers for parallel processing.
- **verbose** (*int, default=1*) – verbosity level
- **fail** (*bool, default=False*) – if True, raises an exception if anything size fails to load.

Returns

a list of "bad" image dictionaries where the size could not be determined. Typically these are corrupted images and should be removed.

Return type List[dict]

Example

```
>>> # Normal case
>>> self = CocoDataset.demo()
>>> bad_imgs = self._ensure_imgsize()
>>> assert len(bad_imgs) == 0
>>> assert self.imgs[1]['width'] == 512
>>> assert self.imgs[2]['width'] == 300
>>> assert self.imgs[3]['width'] == 256
```

```
>>> # Fail cases
>>> self = CocoDataset()
>>> self.add_image('does-not-exist.jpg')
>>> bad_imgs = self._ensure_imgsize()
>>> assert len(bad_imgs) == 1
>>> import pytest
>>> with pytest.raises(Exception):
>>>     self._ensure_imgsize(fail=True)
```

_ensure_image_data(*self*, *gids=None*, *verbose=1*)
Download data from "url" fields if specified.

Parameters **gids** (*List*) – subset of images to download

missing_images(*self*, *check_aux=False*, *verbose=0*)
Check for images that don't exist

Parameters

- **check_aux** (*bool, default=False*) – if specified also checks auxiliary images
- **verbose** (*int*) – verbosity level

Returns bad indexes and paths and ids

Return type List[Tuple[int, str, int]]

corrupted_images(*self*, *check_aux=False*, *verbose=0*)
Check for images that don't exist or can't be opened

Parameters

- **check_aux** (*bool, default=False*) – if specified also checks auxiliary images
- **verbose** (*int*) – verbosity level

Returns bad indexes and paths and ids

Return type List[Tuple[int, str, int]]

rename_categories(*self*, *mapper*, *rebuild=True*, *merge_policy='ignore'*)
Rename categories with a potentially coarser categorization.

Parameters

- **mapper** (*dict or Function*) – maps old names to new names. If multiple names are mapped to the same category, those categories will be merged.
- **merge_policy** (*str*) – How to handle multiple categories that map to the same name. Can be update or ignore.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self.rename_categories({'astronomer': 'person',
>>>                         'astronaut': 'person',
>>>                         'mouth': 'person',
>>>                         'helmet': 'hat'})
>>> assert 'hat' in self.name_to_cat
>>> assert 'helmet' not in self.name_to_cat
>>> # Test merge case
>>> self = kwcoco.CocoDataset.demo()
>>> mapper = {
>>>     'helmet': 'rocket',
>>>     'astronomer': 'rocket',
>>>     'human': 'rocket',
>>>     'mouth': 'helmet',
>>>     'star': 'gas'
>>> }
>>> self.rename_categories(mapper)
```

_ensure_json_serializable(*self*)

_aspycoco(*self*)

reroot(*self*, *new_root=None*, *old_prefix=None*, *new_prefix=None*, *absolute=False*, *check=True*, *safe=True*, *smart=False*)

Rebase image/data paths onto a new image/data root.

Parameters

- **new_root** (*str, default=None*) – New image root. If unspecified the current `self.bundle_dpath` is used. If old_prefix and new_prefix are unspecified, they will attempt to be determined based on the current root (which assumes the file paths exist at that root) and this new root.
- **old_prefix** (*str, default=None*) – If specified, removes this prefix from file names. This also prevents any inferences that might be made via "new_root".
- **new_prefix** (*str, default=None*) – If specified, adds this prefix to the file names. This also prevents any inferences that might be made via "new_root".
- **absolute** (*bool, default=False*) – if True, file names are stored as absolute paths, otherwise they are relative to the new image root.
- **check** (*bool, default=True*) – if True, checks that the images all exist.
- **safe** (*bool, default=True*) – if True, does not overwrite values until all checks pass
- **smart** (*bool, default=False*) – If True, we can try different reroot strategies and choose the one that works. Note, always be wary when algorithms try to be smart.

CommandLine: xdoctest -m kwcoco.coco_dataset MixinCocoExtras.reroot

---

Todo:

- [ ] Incorporate maximum ordered subtree embedding once completed?

Ignore:

```
>>> # There might not be a way to easily handle the cases that I
>>> # want to here. Might need to discuss this.
>>> from kwcoco.coco_dataset import *  # NOQA
>>> import kwcoco
>>> gname = 'images/foo.png'
>>> remote = '/remote/path'
>>> host = ub.ensure_app_cache_dir('kwcoco/tests/reroot')
>>> fpath = join(host, gname)
>>> ub.ensuredir(dirname(fpath))
>>> # In this test the image exists on the host path
>>> import kwimage
>>> kwimage.imwrite(fpath, np.random.rand(8, 8))
>>> #
>>> cases = {}
>>> # * given absolute paths on current machine
>>> cases['abs_curr'] = kwcoco.CocoDataset.from_image_paths([join(host,␣
↪gname)])
>>> # * given "remote" rooted relative paths on current machine
>>> cases['rel_remoterooted_curr'] = kwcoco.CocoDataset.from_image_
↪paths([gname], bundle_dpath=remote)
>>> # * given "host" rooted relative paths on current machine
>>> cases['rel_hostrooted_curr'] = kwcoco.CocoDataset.from_image_
↪paths([gname], bundle_dpath=host)
>>> # * given unrooted relative paths on current machine
>>> cases['rel_unrooted_curr'] = kwcoco.CocoDataset.from_image_
↪paths([gname])
>>> # * given absolute paths on another machine
>>> cases['abs_remote'] = kwcoco.CocoDataset.from_image_paths([join(remote,␣
↪gname)])
>>> def report(dset, name):
>>>     gid = 1
>>>     rel_fpath = dset.index.imgs[gid]['file_name']
>>>     abs_fpath = dset.get_image_fpath(gid)
>>>     color = 'green' if exists(abs_fpath) else 'red'
>>>     print('   * strategy_name = {!r}'.format(name))
>>>     print('       * rel_fpath = {!r}'.format(rel_fpath))
>>>     print('       * ' + ub.color_text('abs_fpath = {!r}'.format(abs_
↪fpath), color))
>>> for key, dset in cases.items():
>>>     print('----')
>>>     print('case key = {!r}'.format(key))
>>>     print('ORIG = {!r}'.format(dset.index.imgs[1]['file_name']))
>>>     print('dset.bundle_dpath = {!r}'.format(dset.bundle_dpath))
>>>     print('missing_gids = {!r}'.format(dset.missing_images()))
>>>     print('cwd = {!r}'.format(os.getcwd()))
>>>     print('host = {!r}'.format(host))
>>>     print('remote = {!r}'.format(remote))
>>>     #
>>>     dset_None_rel = dset.copy().reroot(absolute=False, check=0)
>>>     report(dset_None_rel, 'dset_None_rel')
>>>     #
```

(continues on next page)

(continued from previous page)

```
>>>     dset_None_abs = dset.copy().reroot(absolute=True, check=0)
>>>     report(dset_None_abs, 'dset_None_abs')
>>>     #
>>>     dset_host_rel = dset.copy().reroot(host, absolute=False, check=0)
>>>     report(dset_host_rel, 'dset_host_rel')
>>>     #
>>>     dset_host_abs = dset.copy().reroot(host, absolute=True, check=0)
>>>     report(dset_host_abs, 'dset_host_abs')
>>>     #
>>>     dset_remote_rel = dset.copy().reroot(host, old_prefix=remote,␣
↪absolute=False, check=0)
>>>     report(dset_remote_rel, 'dset_remote_rel')
>>>     #
>>>     dset_remote_abs = dset.copy().reroot(host, old_prefix=remote,␣
↪absolute=True, check=0)
>>>     report(dset_remote_abs, 'dset_remote_abs')
```

Example

```
>>> import kwcoco
>>> def report(dset, name):
>>>     gid = 1
>>>     abs_fpath = dset.get_image_fpath(gid)
>>>     rel_fpath = dset.index.imgs[gid]['file_name']
>>>     color = 'green' if exists(abs_fpath) else 'red'
>>>     print('strategy_name = {!r}'.format(name))
>>>     print(ub.color_text('abs_fpath = {!r}'.format(abs_fpath), color))
>>>     print('rel_fpath = {!r}'.format(rel_fpath))
>>> dset = self = kwcoco.CocoDataset.demo()
>>> # Change base relative directory
>>> bundle_dpath = ub.expandpath('~')
>>> print('ORIG self.imgs = {!r}'.format(self.imgs))
>>> print('ORIG dset.bundle_dpath = {!r}'.format(dset.bundle_dpath))
>>> print('NEW bundle_dpath       = {!r}'.format(bundle_dpath))
>>> self.reroot(bundle_dpath)
>>> report(self, 'self')
>>> print('NEW self.imgs = {!r}'.format(self.imgs))
>>> assert self.imgs[1]['file_name'].startswith('.cache')
```

```
>>> # Use absolute paths
>>> self.reroot(absolute=True)
>>> assert self.imgs[1]['file_name'].startswith(bundle_dpath)
```

```
>>> # Switch back to relative paths
>>> self.reroot()
>>> assert self.imgs[1]['file_name'].startswith('.cache')
```

Example

```
>>> # demo with auxiliary data
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8', aux=True)
>>> bundle_dpath = ub.expandpath('~')
>>> print(self.imgs[1]['file_name'])
>>> print(self.imgs[1]['auxiliary'][0]['file_name'])
>>> self.reroot(new_root=bundle_dpath)
>>> print(self.imgs[1]['file_name'])
>>> print(self.imgs[1]['auxiliary'][0]['file_name'])
>>> assert self.imgs[1]['file_name'].startswith('.cache')
>>> assert self.imgs[1]['auxiliary'][0]['file_name'].startswith('.cache')
```

property data_root(*self*)
In the future we will deprecate data_root for bundle_dpath

property img_root(*self*)
In the future we will deprecate img_root for bundle_dpath

property data_fpath(*self*)
data_fpath is an alias of fpath

class kwcoco.coco_dataset.**MixinCocoObjects**
Bases: object

Expose methods to construct object lists / groups.

This is an alternative vectorized ORM-like interface to the coco dataset

annots(*self*, *aids=None*, *gid=None*, *trackid=None*)
Return vectorized annotation objects

Parameters

- **aids** (*List[int]*) – annotation ids to reference, if unspecified all annotations are returned.
- **gid** (*int*) – return all annotations that belong to this image id. mutually exclusive with other arguments.
- **trackid** (*int*) – return all annotations that belong to this track. mutually exclusive with other arguments.

Returns vectorized annotation object

Return type *Annots*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> annots = self.annots()
>>> print(annots)
<Annots(num=11)>
>>> sub_annots = annots.take([1, 2, 3])
>>> print(sub_annots)
<Annots(num=3)>
>>> print(ub.repr2(sub_annots.get('bbox', None)))
```

(continues on next page)

(continued from previous page)

```
[
    [350, 5, 130, 290],
    None,
    None,
]
```

images(*self*, *gids=None*, *vidid=None*)
Return vectorized image objects

Parameters

- **gids** (*List[int]*) – image ids to reference, if unspecified all images are returned.
- **vidid** (*int*) – returns all images that belong to this video id. mutually exclusive with *gids* arg.

Returns vectorized image object

Return type *Images*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> images = self.images()
>>> print(images)
<Images(num=3)>
```

```
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> vidid = 1
>>> images = self.images(vidid=vidid)
>>> assert all(v == vidid for v in images.lookup('video_id'))
>>> print(images)
<Images(num=2)>
```

categories(*self*, *cids=None*)
Return vectorized category objects

Parameters **cids** (*List[int]*) – category ids to reference, if unspecified all categories are returned.

Returns vectorized category object

Return type *Categories*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> categories = self.categories()
>>> print(categories)
<Categories(num=8)>
```

videos(*self*, *vidids=None*)
Return vectorized video objects

Parameters **vidids** (*List[int]*) – video ids to reference, if unspecified all videos are returned.

Returns vectorized video object

Return type *Videos*

Todo:

- **[ ] This conflicts with what should be the property that** should redirect to `index.videos`, we should resolve this somehow. E.g. all other main members of the index (anns, imgs, cats) have a toplevel dataset property, we don't have one for videos because the name we would pick conflicts with this.

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> videos = self.videos()
>>> print(videos)
>>> videos.lookup('name')
>>> videos.lookup('id')
>>> print('videos.objs = {}'.format(ub.repr2(videos.objs[0:2], nl=1)))
```

class kwcoco.coco_dataset.**MixinCocoStats**

Bases: `object`

Methods for getting stats about the dataset

property n_annots(*self*)

property n_images(*self*)

property n_cats(*self*)

property n_videos(*self*)

keypoint_annotation_frequency(*self*)

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo('shapes', rng=0)
>>> hist = self.keypoint_annotation_frequency()
>>> hist = ub.odict(sorted(hist.items()))
>>> # FIXME: for whatever reason demodata generation is not determenistic when␣
↪seeded
>>> print(ub.repr2(hist))  # xdoc: +IGNORE_WANT
{
    'bot_tip': 6,
    'left_eye': 14,
    'mid_tip': 6,
    'right_eye': 14,
    'top_tip': 6,
}
```

category_annotation_frequency(*self*)
Reports the number of annotations of each category

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> hist = self.category_annotation_frequency()
>>> print(ub.repr2(hist))
{
    'astroturf': 0,
    'human': 0,
    'astronaut': 1,
    'astronomer': 1,
    'helmet': 1,
    'rocket': 1,
    'mouth': 2,
    'star': 5,
}
```

category_annotation_type_frequency(*self*)
Reports the number of annotations of each type for each category

Example

```
>>> self = CocoDataset.demo()
>>> hist = self.category_annotation_frequency()
>>> print(ub.repr2(hist))
```

conform(*self*, ***config*)
Make the COCO file conform a stricter spec, infers attibutes where possible.

Corresponds to the `kwcoco conform` CLI tool.

KWArgs:

****config :** pycocotools_info (default=True): returns info required by pycocotools ensure_imgsize (default=True): ensure image size is populated legacy (default=False): if true tries to convert data structures to items compatible with the original pycocotools spec

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> dset.index.imgs[1].pop('width')
>>> dset.conform(legacy=True)
>>> assert 'width' in dset.index.imgs[1]
>>> assert 'area' in dset.index.anns[1]
```

validate(*self*, ***config*)
Performs checks on this coco dataset.

Corresponds to the `kwcoco validate` CLI tool.

KWArgs:

****config :** schema (default=True): validates the json-schema unique (default=True): validates unique secondary keys missing (default=True): validates registered files exist corrupted (default=False): validates data in registered files verbose (default=1): verbosity flag fastfail (default=False): if True raise errors immediately

Returns

result containing keys: status (bool): False if any errors occurred errors (List[str]): list of all error messages missing (List): List of any missing images corrupted (List): List of any corrupted images

Return type dict

SeeAlso: **:method:`_check_integrity`** - performs internal checks

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> import pytest
>>> with pytest.warns(UserWarning):
>>>     result = self.validate()
>>> assert not result['errors']
>>> assert result['warnings']
```

stats(*self*, ***kwargs*)

This function corresponds to **:module:`kwcoco.cli.coco_stats`**.

KWargs: basic(bool, default=True): return basic stats' extended(bool, default=True): return extended stats' catfreq(bool, default=True): return category frequency stats' boxes(bool, default=False): return bounding box stats'

annot_attrs(bool, default=True): return annotation attribute information' image_attrs(bool, default=True): return image attribute information'

Returns info

Return type dict

basic_stats(*self*)

Reports number of images, annotations, and categories.

SeeAlso: *basic_stats() extended_stats()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> print(ub.repr2(self.basic_stats()))
{
    'n_anns': 11,
    'n_imgs': 3,
    'n_videos': 0,
    'n_cats': 8,
}
```

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> dset = random_video_dset(render=True, num_frames=2, num_tracks=10, rng=0)
>>> print(ub.repr2(dset.basic_stats()))
{
    'n_anns': 20,
    'n_imgs': 2,
    'n_videos': 1,
    'n_cats': 3,
}
```

extended_stats(*self*)

Reports number of images, annotations, and categories.

SeeAlso: *basic_stats() extended_stats()*

Example

```
>>> self = CocoDataset.demo()
>>> print(ub.repr2(self.extended_stats()))
```

boxsize_stats(*self*, *anchors=None*, *perclass=True*, *gids=None*, *aids=None*, *verbose=0*, *clusterkw={}*, *statskw={}*)

Compute statistics about bounding box sizes.

Also computes anchor boxes using kmeans if `anchors` is specified.

Parameters

- **anchors** (*int*) – if specified also computes box anchors via KMeans clustering
- **perclass** (*bool*) – if True also computes stats for each category
- **gids** (*List[int], default=None*) – if specified only compute stats for these image ids.
- **aids** (*List[int], default=None*) – if specified only compute stats for these annotation ids.
- **verbose** (*int*) – verbosity level
- **clusterkw** (*dict*) – kwargs for `sklearn.cluster.KMeans` used if computing anchors.
- **statskw** (*dict*) – kwargs for `kwarray.stats_dict()`

Returns Stats are returned in width-height format.

Return type Dict[str, Dict[str, Dict | ndarray]

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes32')
>>> infos = self.boxsize_stats(anchors=4, perclass=False)
>>> print(ub.repr2(infos, nl=-1, precision=2))
```

```
>>> infos = self.boxsize_stats(gids=[1], statskw=dict(median=True))
>>> print(ub.repr2(infos, nl=-1, precision=2))
```

find_representative_images(*self*, *gids=None*)

Find images that have a wide array of categories. Attempt to find the fewest images that cover all categories using images that contain both a large and small number of annotations.

Parameters **gids** (*None | List*) – Subset of image ids to consider when finding representative images. Uses all images if unspecified.

Returns list of image ids determined to be representative

Return type List

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> gids = self.find_representative_images()
>>> print('gids = {!r}'.format(gids))
>>> gids = self.find_representative_images([3])
>>> print('gids = {!r}'.format(gids))
```

```
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> gids = self.find_representative_images()
>>> print('gids = {!r}'.format(gids))
>>> valid = {7, 1}
>>> gids = self.find_representative_images(valid)
>>> assert valid.issuperset(gids)
>>> print('gids = {!r}'.format(gids))
```

class kwcoco.coco_dataset.**_NextId**(*parent*)

Bases: object

Helper class to tracks unused ids for new items

_update_unused(*self*, *key*)

Scans for what the next safe id can be for `key`

get(*self*, *key*)

Get the next safe item id for `key`

class kwcoco.coco_dataset.**_ID_Remapper**(*reuse=False*)

Bases: object

Helper to recycle ids for unions.

For each dataset we create a mapping between each old id and a new id. If possible and reuse=True we allow the new id to match the old id. After each dataset is finished we mark all those ids as used and subsequent new-ids cannot be chosen from that pool.

Parameters **reuse** (*bool*) – if True we are allowed to reuse ids as long as they haven't been used before.

Example

```
>>> video_trackids = [[1, 1, 3, 3, 200, 4], [204, 1, 2, 3, 3, 4, 5, 9]]
>>> self = _ID_Remapper(reuse=True)
>>> for tids in video_trackids:
>>>     new_tids = [self.remap(old_tid) for old_tid in tids]
>>>     self.block_seen()
>>>     print('new_tids = {!r}'.format(new_tids))
new_tids = [1, 1, 3, 3, 200, 4]
new_tids = [204, 205, 2, 206, 206, 207, 5, 9]
>>> #
>>> self = _ID_Remapper(reuse=False)
>>> for tids in video_trackids:
>>>     new_tids = [self.remap(old_tid) for old_tid in tids]
>>>     self.block_seen()
>>>     print('new_tids = {!r}'.format(new_tids))
new_tids = [0, 0, 1, 1, 2, 3]
new_tids = [4, 5, 6, 7, 7, 8, 9, 10]
```

remap(*self*, *old_id*)
Convert a old-id into a new-id. If self.reuse is True then we will return the same id if it hasn't been blocked yet.

block_seen(*self*)
Mark all seen ids as unable to be used. Any ids sent to remap will now generate new ids.

next_id(*self*)
Generate a new id that hasnt been used yet

class kwcoco.coco_dataset.**UniqueNameRemapper**
Bases: object

helper to ensure names will be unique by appending suffixes

Example

```
>>> from kwcoco.coco_dataset import *  # NOQA
>>> self = UniqueNameRemapper()
>>> assert self.remap('foo') == 'foo'
>>> assert self.remap('foo') == 'foo_v001'
>>> assert self.remap('foo') == 'foo_v002'
>>> assert self.remap('foo_v001') == 'foo_v003'
```

remap(*self*, *name*)

class kwcoco.coco_dataset.**MixinCocoDraw**
Bases: object

Matplotlib / display functionality

imread(*self*, *gid*)
Loads a particular image

draw_image(*self*, *gid*, *channels=None*)
Use kwimage to draw all annotations on an image and return the pixels as a numpy array.

Returns canvas

Return type ndarray

SeeAlso *draw_image() show_image()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes8')
>>> self.draw_image(1)
>>> # Now you can dump the annotated image to disk / whatever
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(canvas)
```

show_image(*self*, *gid=None*, *aids=None*, *aid=None*, *channels=None*, ***kwargs*)
Use matplotlib to show an image with annotations overlaid

Parameters

- **gid** (*int*) – image to show
- **aids** (*list*) – aids to highlight within the image
- **aid** (*int*) – a specific aid to focus on. If gid is not give, look up gid based on this aid.
- ****kwargs** – show_annots, show_aid, show_catname, show_kpname, show_segmentation, title, show_gid, show_filename, show_boxes,

SeeAlso *draw_image() show_image()*

Ignore: # Programatically collect the kwargs for docs generation import xinspect import kwcoco kwargs = xinspect.get_kwargs(kwcoco.CocoDataset.show_image) print(ub.repr2(list(kwargs.keys()), nl=1, si=1))

class kwcoco.coco_dataset.**MixinCocoAddRemove**
Bases: object

Mixin functions to dynamically add / remove annotations images and categories while maintaining lookup indexes.

add_video(*self*, *name*, *id=None*, ***kw*)
Add a video to the dataset (dynamically updates the index)

Parameters

- **name** (*str*) – Unique name for this video.
- **id** (*None or int*) – ADVANCED. Force using this image id.
- ****kw** – stores arbitrary key/value pairs in this new video

Returns the video id assigned to the new video

Return type int

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset()
>>> print('self.index.videos = {}'.format(ub.repr2(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.repr2(self.index.imgs, nl=1)))
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

```
>>> vidid1 = self.add_video('foo', id=3)
>>> vidid2 = self.add_video('bar')
>>> vidid3 = self.add_video('baz')
>>> print('self.index.videos = {}'.format(ub.repr2(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.repr2(self.index.imgs, nl=1)))
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

```
>>> gid1 = self.add_image('foo1.jpg', video_id=vidid1, frame_index=0)
>>> gid2 = self.add_image('foo2.jpg', video_id=vidid1, frame_index=1)
>>> gid3 = self.add_image('foo3.jpg', video_id=vidid1, frame_index=2)
>>> gid4 = self.add_image('bar1.jpg', video_id=vidid2, frame_index=0)
>>> print('self.index.videos = {}'.format(ub.repr2(self.index.videos, nl=1)))
>>> print('self.index.imgs = {}'.format(ub.repr2(self.index.imgs, nl=1)))
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

```
>>> self.remove_images([gid2])
>>> print('self.index.vidid_to_gids = {!r}'.format(self.index.vidid_to_gids))
```

add_image(*self*, *file_name=None*, *id=None*, ***kw*)

Add an image to the dataset (dynamically updates the index)

Parameters

- **file_name** (*str*) – relative or absolute path to image
- **id** (*None or int*) – ADVANCED. Force using this image id.
- **name** (*str*) – a unique key to identify this image
- **width** (*int*) – base width of the image
- **height** (*int*) – base height of the image
- **channels** (*ChannelSpec*) – specification of base channels
- **auxiliary** (*List[Dict]*) – specification of auxiliary information
- **video_id** (*int*) – parent video, if applicable
- **frame_index** (*int*) – frame index in parent video
- **timestamp** (*number | str*) – timestamp of frame index
- ****kw** – stores arbitrary key/value pairs in this new image

Returns the image id assigned to the new image

Return type int

SeeAlso: *add_image() add_images() ensure_image()*

Example

```
>>> self = CocoDataset.demo()
>>> import kwimage
>>> gname = kwimage.grab_test_image_fpath('paraview')
>>> gid = self.add_image(gname)
>>> assert self.imgs[gid]['file_name'] == gname
```

add_annotation(*self*, *image_id*, *category_id=None*, *bbox=ub.NoParam*, *segmentation=ub.NoParam*, *keypoints=ub.NoParam*, *id=None*, ***kw*)

Add an annotation to the dataset (dynamically updates the index)

Parameters

- **image_id** (*int*) – image_id the annoatation is added to.
- **category_id** (*int | None*) – category_id for the new annotaiton
- **bbox** (*list | kwimage.Boxes*) – bounding box in xywh format
- **segmentation** (*MaskLike | MultiPolygonLike*) – keypoints in some accepted format, see **:method:`kwimage.Mask.to_coco`** and **:method:`kwimage.MultiPolygon.to_coco`**.
- **keypoints** (*KeypointsLike*) – keypoints in some accepted format, see **:method:`kwimage.Keypoints.to_coco`**..
- **id** (*None | int*) – Force using this annotation id. Typically you should NOT specify this. A new unused id will be chosen and returned.
- ****kw** – stores arbitrary key/value pairs in this new image, Common respected key/values include but are not limited to the following:

  track_id (int | str): some value used to associate annotations that belong to the same "track".

  score' : float

  prob' : List[float]

  weight (float): a weight, usually used to indicate if a ground truth annotation is difficult / important. This generalizes standard "is_hard" or "ignore" attributes in other formats.

  caption (str): a text caption for this annotation

Returns the annotation id assigned to the new annotation

Return type int

SeeAlso: *add_annotation() add_annotations()*

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> image_id = 1
>>> cid = 1
>>> bbox = [10, 10, 20, 20]
>>> aid = self.add_annotation(image_id, cid, bbox)
>>> assert self.anns[aid]['bbox'] == bbox
```

Example

```
>>> import kwimage
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> new_det = kwimage.Detections.random(1, segmentations=True, keypoints=True)
>>> # kwimage datastructures have methods to convert to coco recognized formats
>>> new_ann_data = list(new_det.to_coco(style='new'))[0]
>>> image_id = 1
>>> aid = self.add_annotation(image_id, **new_ann_data)
>>> # Lookup the annotation we just added
>>> ann = self.index.anns[aid]
>>> print('ann = {}'.format(ub.repr2(ann, nl=-2)))
```

Example

```
>>> # Attempt to add annot without a category or bbox
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> image_id = 1
>>> aid = self.add_annotation(image_id)
>>> assert None in self.index.cid_to_aids
```

Example

```
>>> # Attempt to add annot using various styles of kwimage structures
>>> import kwcoco
>>> import kwimage
>>> self = kwcoco.CocoDataset.demo()
>>> image_id = 1
>>> #--
>>> kw = {}
>>> kw['segmentation'] = kwimage.Polygon.random()
>>> kw['keypoints'] = kwimage.Points.random()
>>> aid = self.add_annotation(image_id, **kw)
>>> ann = self.index.anns[aid]
>>> print('ann = {}'.format(ub.repr2(ann, nl=2)))
>>> #--
```

(continues on next page)

(continued from previous page)

```
>>> kw = {}
>>> kw['segmentation'] = kwimage.Mask.random()
>>> aid = self.add_annotation(image_id, **kw)
>>> ann = self.index.anns[aid]
>>> assert ann.get('segmentation', None) is not None
>>> print('ann = {}'.format(ub.repr2(ann, nl=2)))
>>> #--
>>> kw = {}
>>> kw['segmentation'] = kwimage.Mask.random().to_array_rle()
>>> aid = self.add_annotation(image_id, **kw)
>>> ann = self.index.anns[aid]
>>> assert ann.get('segmentation', None) is not None
>>> print('ann = {}'.format(ub.repr2(ann, nl=2)))
>>> #--
>>> kw = {}
>>> kw['segmentation'] = kwimage.Polygon.random().to_coco()
>>> kw['keypoints'] = kwimage.Points.random().to_coco()
>>> aid = self.add_annotation(image_id, **kw)
>>> ann = self.index.anns[aid]
>>> assert ann.get('segmentation', None) is not None
>>> assert ann.get('keypoints', None) is not None
>>> print('ann = {}'.format(ub.repr2(ann, nl=2)))
```

add_category(*self*, *name*, *supercategory=None*, *id=None*, ***kw*)

Adds a category

Parameters

- **name** (*str*) – name of the new category
- **supercategory** (*str, optional*) – parent of this category
- **id** (*int, optional*) – use this category id, if it was not taken
- ****kw** – stores arbitrary key/value pairs in this new image

Returns the category id assigned to the new category

Return type int

SeeAlso: *add_category() ensure_category()*

Example

```
>>> self = CocoDataset.demo()
>>> prev_n_cats = self.n_cats
>>> cid = self.add_category('dog', supercategory='object')
>>> assert self.cats[cid]['name'] == 'dog'
>>> assert self.n_cats == prev_n_cats + 1
>>> import pytest
>>> with pytest.raises(ValueError):
>>>     self.add_category('dog', supercategory='object')
```

ensure_image(*self*, *file_name*, *id=None*, ***kw*)

Like *add_image()*,, but returns the existing image id if it already exists instead of failing. In this case all metadata is ignored.

Parameters

- **file_name** (*str*) – relative or absolute path to image
- **id** (*None or int*) – ADVANCED. Force using this image id.
- ****kw** – stores arbitrary key/value pairs in this new image

Returns the existing or new image id

Return type int

SeeAlso: *add_image() add_images() ensure_image()*

ensure_category(*self*, *name*, *supercategory=None*, *id=None*, ***kw*)

Like *add_category()*, but returns the existing category id if it already exists instead of failing. In this case all metadata is ignored.

Returns the existing or new category id

Return type int

SeeAlso: *add_category() ensure_category()*

add_annotations(*self*, *anns*)

Faster less-safe multi-item alternative to add_annotation.

We assume the annotations are well formatted in kwcoco compliant dictionaries, including the "id" field. No validation checks are made when calling this function.

Parameters **anns** (*List[Dict]*) – list of annotation dictionaries

SeeAlso: *add_annotation() add_annotations()*

Example

```
>>> self = CocoDataset.demo()
>>> anns = [self.anns[aid] for aid in [2, 3, 5, 7]]
>>> self.remove_annotations(anns)
>>> assert self.n_annots == 7 and self._check_index()
>>> self.add_annotations(anns)
>>> assert self.n_annots == 11 and self._check_index()
```

add_images(*self*, *imgs*)

Faster less-safe multi-item alternative

We assume the images are well formatted in kwcoco compliant dictionaries, including the "id" field. No validation checks are made when calling this function.

Note: THIS FUNCTION WAS DESIGNED FOR SPEED, AS SUCH IT DOES NOT CHECK IF THE IMAGE-IDs or FILE_NAMES ARE DUPLICATED AND WILL BLINDLY ADD DATA EVEN IF IT IS BAD. THE SINGLE IMAGE VERSION IS SLOWER BUT SAFER.

Parameters **imgs** (*List[Dict]*) – list of image dictionaries

SeeAlso: *add_image() add_images() ensure_image()*

Example

```
>>> imgs = CocoDataset.demo().dataset['images']
>>> self = CocoDataset()
>>> self.add_images(imgs)
>>> assert self.n_images == 3 and self._check_index()
```

clear_images(*self*)
Removes all images and annotations (but not categories)

Example

```
>>> self = CocoDataset.demo()
>>> self.clear_images()
>>> print(ub.repr2(self.basic_stats(), nobr=1, nl=0, si=1))
n_anns: 0, n_imgs: 0, n_videos: 0, n_cats: 8
```

clear_annotations(*self*)
Removes all annotations (but not images and categories)

Example

```
>>> self = CocoDataset.demo()
>>> self.clear_annotations()
>>> print(ub.repr2(self.basic_stats(), nobr=1, nl=0, si=1))
n_anns: 0, n_imgs: 3, n_videos: 0, n_cats: 8
```

remove_annotation(*self*, *aid_or_ann*)
Remove a single annotation from the dataset

If you have multiple annotations to remove its more efficient to remove them in batch with `self.remove_annotations`

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> aids_or_anns = [self.anns[2], 3, 4, self.anns[1]]
>>> self.remove_annotations(aids_or_anns)
>>> assert len(self.dataset['annotations']) == 7
>>> self._check_index()
```

remove_annotations(*self*, *aids_or_anns*, *verbose=0*, *safe=True*)
Remove multiple annotations from the dataset.

Parameters

- **anns_or_aids** (*List*) – list of annotation dicts or ids
- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns num_removed: information on the number of items removed

Return type Dict

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> prev_n_annots = self.n_annots
>>> aids_or_anns = [self.anns[2], 3, 4, self.anns[1]]
>>> self.remove_annotations(aids_or_anns)  # xdoc: +IGNORE_WANT
{'annotations': 4}
>>> assert len(self.dataset['annotations']) == prev_n_annots - 4
>>> self._check_index()
```

remove_categories(*self*, *cat_identifiers*, *keep_annots=False*, *verbose=0*, *safe=True*)

Remove categories and all annotations in those categories. Currently does not change any hierarchy information

Parameters

- **cat_identifiers** (*List*) – list of category dicts, names, or ids
- **keep_annots** (*bool, default=False*) – if True, keeps annotations, but removes category labels.
- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns num_removed: information on the number of items removed

Return type Dict

Example

```
>>> self = CocoDataset.demo()
>>> cat_identifiers = [self.cats[1], 'rocket', 3]
>>> self.remove_categories(cat_identifiers)
>>> assert len(self.dataset['categories']) == 5
>>> self._check_index()
```

remove_images(*self*, *gids_or_imgs*, *verbose=0*, *safe=True*)

Remove images and any annotations contained by them

Parameters

- **gids_or_imgs** (*List*) – list of image dicts, names, or ids
- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns num_removed: information on the number of items removed

Return type Dict

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> assert len(self.dataset['images']) == 3
>>> gids_or_imgs = [self.imgs[2], 'astro.png']
>>> self.remove_images(gids_or_imgs)  # xdoc: +IGNORE_WANT
{'annotations': 11, 'images': 2}
>>> assert len(self.dataset['images']) == 1
>>> self._check_index()
>>> gids_or_imgs = [3]
>>> self.remove_images(gids_or_imgs)
>>> assert len(self.dataset['images']) == 0
>>> self._check_index()
```

remove_videos(*self*, *vidids_or_videos*, *verbose=0*, *safe=True*)

Remove videos and any images / annotations contained by them

Parameters

- **vidids_or_videos** (*List*) – list of video dicts, names, or ids
- **safe** (*bool, default=True*) – if True, we perform checks to remove duplicates and non-existing identifiers.

Returns num_removed: information on the number of items removed

Return type Dict

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo('vidshapes8')
>>> assert len(self.dataset['videos']) == 8
>>> vidids_or_videos = [self.dataset['videos'][0]['id']]
>>> self.remove_videos(vidids_or_videos)  # xdoc: +IGNORE_WANT
{'annotations': 4, 'images': 2, 'videos': 1}
>>> assert len(self.dataset['videos']) == 7
>>> self._check_index()
```

remove_annotation_keypoints(*self*, *kp_identifiers*)

Removes all keypoints with a particular category

Parameters **kp_identifiers** (*List*) – list of keypoint category dicts, names, or ids

Returns num_removed: information on the number of items removed

Return type Dict

remove_keypoint_categories(*self*, *kp_identifiers*)

Removes all keypoints of a particular category as well as all annotation keypoints with those ids.

Parameters **kp_identifiers** (*List*) – list of keypoint category dicts, names, or ids

Returns num_removed: information on the number of items removed

Return type Dict

Example

```
>>> self = CocoDataset.demo('shapes', rng=0)
>>> kp_identifiers = ['left_eye', 'mid_tip']
>>> remove_info = self.remove_keypoint_categories(kp_identifiers)
>>> print('remove_info = {!r}'.format(remove_info))
>>> # FIXME: for whatever reason demodata generation is not determenistic when␣
↪seeded
>>> # assert remove_info == {'keypoint_categories': 2, 'annotation_keypoints': 16,
↪'reflection_ids': 1}
>>> assert self._resolve_to_kpcat('right_eye')['reflection_id'] is None
```

set_annotation_category(*self*, *aid_or_ann*, *cid_or_cat*)

Sets the category of a single annotation

Parameters

- **aid_or_ann** (*dict | int*) – annotation dict or id
- **cid_or_cat** (*dict | int*) – category dict or id

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> old_freq = self.category_annotation_frequency()
>>> aid_or_ann = aid = 2
>>> cid_or_cat = new_cid = self.ensure_category('kitten')
>>> self.set_annotation_category(aid, new_cid)
>>> new_freq = self.category_annotation_frequency()
>>> print('new_freq = {}'.format(ub.repr2(new_freq, nl=1)))
>>> print('old_freq = {}'.format(ub.repr2(old_freq, nl=1)))
>>> assert sum(new_freq.values()) == sum(old_freq.values())
>>> assert new_freq['kitten'] == 1
```

class kwcoco.coco_dataset.**CocoIndex**(*index*)

Bases: `object`

Fast lookup index for the COCO dataset with dynamic modification

Variables

- **imgs** (*Dict[int, dict]*) – mapping between image ids and the image dictionaries
- **anns** (*Dict[int, dict]*) – mapping between annotation ids and the annotation dictionaries
- **cats** (*Dict[int, dict]*) – mapping between category ids and the category dictionaries

_set

__nonzero__

_set_sorted_by_frame_index(*index*, *gids=None*)

Helper for ensuring that vidid_to_gids returns image ids ordered by frame index.

__bool__(*index*)

property cid_to_gids(*index*)

Example

```
>>> import kwcoco
>>> self = dset = kwcoco.CocoDataset()
>>> self.index.cid_to_gids
```

_add_video(*index*, *vidid*, *video*)

_add_image(*index*, *gid*, *img*)

Example

```
>>> # Test adding image to video that doesnt exist
>>> import kwcoco
>>> self = dset = kwcoco.CocoDataset()
>>> dset.add_image(file_name='frame1', video_id=1, frame_index=0)
>>> dset.add_image(file_name='frame2', video_id=1, frame_index=0)
>>> dset._check_pointers()
>>> dset._check_index()
>>> print('dset.index.vidid_to_gids = {!r}'.format(dset.index.vidid_to_gids))
>>> assert len(dset.index.vidid_to_gids) == 1
>>> dset.add_video(name='foo-vid', id=1)
>>> assert len(dset.index.vidid_to_gids) == 1
>>> dset._check_pointers()
>>> dset._check_index()
```

_add_images(*index*, *imgs*)

Note: THIS FUNCTION WAS DESIGNED FOR SPEED, AS SUCH IT DOES NOT CHECK IF THE IMAGE-IDs or FILE_NAMES ARE DUPLICATED AND WILL BLINDLY ADD DATA EVEN IF IT IS BAD. THE SINGLE IMAGE VERSION IS SLOWER BUT SAFER.

Ignore: # If we did do checks, what would be the fastest way?

import kwcoco x = kwcoco.CocoDataset() for i in range(1000):

x.add_image(file_name=str(i))

y = kwcoco.CocoDataset() for i in range(1000, 2000):

y.add_image(file_name=str(i))

imgs = list(y.imgs.values()) new_file_name_to_img = {img[‘file_name’]: img for img in imgs}

import ubelt as ub ti = ub.Timerit(100, bestof=10, verbose=2)

for timer in ti.reset(‘set intersection’):

with timer: # WINNER bool(set(x.index.file_name_to_img) & set(new_file_name_to_img))

for timer in ti.reset('dict contains'):

with timer:

any(f in x.index.file_name_to_img for f in new_file_name_to_img.keys())

_add_annotation(*index*, *aid*, *gid*, *cid*, *tid*, *ann*)

_add_annotations(*index*, *anns*)

_add_category(*index*, *cid*, *name*, *cat*)

_remove_all_annotations(*index*)

_remove_all_images(*index*)

_remove_annotations(*index*, *remove_aids*, *verbose=0*)

_remove_categories(*index*, *remove_cids*, *verbose=0*)

_remove_images(*index*, *remove_gids*, *verbose=0*)

_remove_videos(*index*, *remove_vidids*, *verbose=0*)

clear(*index*)

build(*index*, *parent*)

Build all id-to-obj reverse indexes from scratch.

Parameters **parent** (*CocoDataset*) – the dataset to index

Notation: aid - Annotation ID gid - imaGe ID cid - Category ID vidid - Video ID

Example

```
>>> from kwcoco.demo.toydata import *  # NOQA
>>> parent = CocoDataset.demo('vidshapes1', num_frames=4, rng=1)
>>> index = parent.index
>>> index.build(parent)
```

class kwcoco.coco_dataset.**MixinCocoIndex**

Bases: object

Give the dataset top level access to index attributes

property anns(*self*)

property imgs(*self*)

property cats(*self*)

property gid_to_aids(*self*)

property cid_to_aids(*self*)

property name_to_cat(*self*)

class kwcoco.coco_dataset.**CocoDataset**(*data=None*, *tag=None*, *bundle_dpath=None*, *img_root=None*, *fname=None*, *autobuild=True*)

Bases: *kwcoco.abstract_coco_dataset.AbstractCocoDataset*, *MixinCocoAddRemove*, *MixinCocoStats*, *MixinCocoObjects*, *MixinCocoDraw*, *MixinCocoAccessors*, *MixinCocoExtras*, *MixinCocoIndex*, *MixinCocoDepricate*, ubelt.NiceRepr

Notes

A keypoint annotation

{ “image_id” : int, “category_id” : int, “keypoints” : [x1,y1,v1,...,xk,yk,vk], “score” : float,

} Note that `v[i]` is a visibility flag, where v=0: not labeled,

v=1: labeled but not visible, and v=2: labeled and visible.

A bounding box annotation

{ “image_id” : int, “category_id” : int, “bbox” : [x,y,width,height], “score” : float,

}

We also define a non-standard “line” annotation (which our fixup scripts will interpret as the diameter of a circle to convert into a bounding box)

A line* annotation (note this is a non-standard field)

{ “image_id” : int, “category_id” : int, “line” : [x1,y1,x2,y2], “score” : float,

}

Lastly, note that our datasets will sometimes specify multiple bbox, line, and/or, keypoints fields. In this case we may also specify a field roi_shape, which denotes which field is the “main” annotation type.

Variables

- **dataset** (*Dict*) – raw json data structure. This is the base dictionary that contains {‘annotations’: List, ‘images’: List, ‘categories’: List}
- **index** (CocoIndex) – an efficient lookup index into the coco data structure. The index defines its own attributes like `anns`, `cats`, `imgs`, etc. See *CocoIndex* for more details on which attributes are available.
- **fpath** (*PathLike | None*) – if known, this stores the filepath the dataset was loaded from
- **tag** (*str*) – A tag indicating the name of the dataset.
- **bundle_dpath** (*PathLike | None*) – If known, this is the root path that all image file names are relative to. This can also be manually overwritten by the user.
- **hashid** (*str | None*) – If computed, this will be a hash uniquely identifing the dataset. To ensure this is computed see `_build_hashid()`.

References

http://cocodataset.org/#format http://cocodataset.org/#download

CommandLine: python -m kwcoco.coco_dataset CocoDataset –show

Example

```
>>> dataset = demo_coco_data()
>>> self = CocoDataset(dataset, tag='demo')
>>> # xdoctest: +REQUIRES(--show)
>>> self.show_image(gid=2)
>>> from matplotlib import pyplot as plt
>>> plt.show()
```

property fpath(*self*)
In the future we will deprecate img_root for bundle_dpath

_infer_dirs(*self*)

Example

self = dset

classmethod from_data(*CocoDataset*, *data*, *bundle_dpath=None*, *img_root=None*)
Constructor from a json dictionary

classmethod from_image_paths(*CocoDataset*, *gpaths*, *bundle_dpath=None*, *img_root=None*)
Constructor from a list of images paths.

This is a convinience method.

Parameters **gpaths** (*List[str]*) – list of image paths

Example

```
>>> coco_dset = CocoDataset.from_image_paths(['a.png', 'b.png'])
>>> assert coco_dset.n_images == 2
```

classmethod from_coco_paths(*CocoDataset*, *fpaths*, *max_workers=0*, *verbose=1*, *mode='thread'*, *union='try'*)
Constructor from multiple coco file paths.

Loads multiple coco datasets and unions the result

Notes

if the union operation fails, the list of individually loaded files is returned instead.

Parameters

- **fpaths** (*List[str]*) – list of paths to multiple coco files to be loaded and unioned.
- **max_workers** (*int, default=0*) – number of worker threads / processes
- **verbose** (*int*) – verbosity level
- **mode** (*str*) – thread, process, or serial
- **union** (*str | bool, default='try'*) – If True, unions the result datasets after loading. If False, just returns the result list. If 'try', then try to preform the union, but return the result list if it fails.

copy(*self*)
Deep copies this object

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> new = self.copy()
>>> assert new.imgs[1] is new.dataset['images'][0]
>>> assert new.imgs[1] == self.dataset['images'][0]
>>> assert new.imgs[1] is not self.dataset['images'][0]
```

__nice__(*self*)

dumps(*self*, *indent=None*, *newlines=False*)
Writes the dataset out to the json format

Parameters **newlines** (*bool*) – if True, each annotation, image, category gets its own line

Notes

Using newlines=True is similar to: print(ub.repr2(dset.dataset, nl=2, trailsep=False)) However, the above may not output valid json if it contains ndarrays.

Example

```
>>> from kwcoco.coco_dataset import *
>>> import json
>>> self = CocoDataset.demo()
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

dump(*self*, *file*, *indent=None*, *newlines=False*)
Writes the dataset out to the json format

Parameters

- **file** (*PathLike | FileLike*) – Where to write the data. Can either be a path to a file or an open file pointer / stream.
- **newlines** (*bool*) – if True, each annotation, image, category gets its own line.

Example

```
>>> import tempfile
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = CocoDataset(dataset, tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file, newlines=True)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = CocoDataset(dataset, tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

_check_json_serializable(*self*, *verbose=1*)

Debug which part of a coco dataset might not be json serializable

_check_integrity(*self*)

perform all checks

_check_index(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self._check_index()
>>> # Force a failure
>>> self.index.anns.pop(1)
>>> self.index.anns.pop(2)
>>> import pytest
>>> with pytest.raises(AssertionError):
>>>     self._check_index()
```

_check_pointers(*self*, *verbose=1*)

Check that all category and image ids referenced by annotations exist

_build_index(*self*)

union(**others*, *disjoint_tracks=True*, ***kwargs*)

Merges multiple *CocoDataset* items into one. Names and associations are retained, but ids may be different.

Parameters

- ***others** – a series of CocoDatasets that we will merge. Note, if called as an instance method, the "self" instance will be the first item in the "others" list. But if called like a classmethod, "others" will be empty by default.
- **disjoint_tracks** (*bool, default=True*) – if True, we will assume track-ids are disjoint and if two datasets share the same track-id, we will disambiguate them. Otherwise they will be copied over as-is.
- ****kwargs** – constructor options for the new merged CocoDataset

Returns a new merged coco dataset

Return type *CocoDataset*

CommandLine: xdoctest -m kwcoco.coco_dataset CocoDataset.union

Example

```
>>> # Test union works with different keypoint categories
>>> dset1 = CocoDataset.demo('shapes1')
>>> dset2 = CocoDataset.demo('shapes2')
>>> dset1.remove_keypoint_categories(['bot_tip', 'mid_tip', 'right_eye'])
>>> dset2.remove_keypoint_categories(['top_tip', 'left_eye'])
>>> dset_12a = CocoDataset.union(dset1, dset2)
>>> dset_12b = dset1.union(dset2)
>>> dset_21 = dset2.union(dset1)
>>> def add_hist(h1, h2):
>>>     return {k: h1.get(k, 0) + h2.get(k, 0) for k in set(h1) | set(h2)}
>>> kpfreq1 = dset1.keypoint_annotation_frequency()
>>> kpfreq2 = dset2.keypoint_annotation_frequency()
>>> kpfreq_want = add_hist(kpfreq1, kpfreq2)
>>> kpfreq_got1 = dset_12a.keypoint_annotation_frequency()
>>> kpfreq_got2 = dset_12b.keypoint_annotation_frequency()
>>> assert kpfreq_want == kpfreq_got1
>>> assert kpfreq_want == kpfreq_got2
```

```
>>> # Test disjoint gid datasets
>>> import kwcoco
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> for new_gid, img in enumerate(dset1.dataset['images'], start=10):
>>>     for aid in dset1.gid_to_aids[img['id']]:
>>>         dset1.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset1._build_index()
>>> # ------
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> for new_gid, img in enumerate(dset2.dataset['images'], start=100):
>>>     for aid in dset2.gid_to_aids[img['id']]:
```

(continues on next page)

(continued from previous page)

```
>>>         dset2.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset2._build_index()
>>> others = [dset1, dset2]
>>> merged = kwcoco.CocoDataset.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([10, 11, 12, 100, 101]) == set(merged.imgs)
```

```
>>> # Test data is not preserved
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> others = (dset1, dset2)
>>> cls = self = kwcoco.CocoDataset
>>> merged = cls.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([1, 2, 3, 4, 5]) == set(merged.imgs)
```

```
>>> # Test track-ids are mapped correctly
>>> dset1 = kwcoco.CocoDataset.demo('vidshapes1')
>>> dset2 = kwcoco.CocoDataset.demo('vidshapes2')
>>> dset3 = kwcoco.CocoDataset.demo('vidshapes3')
>>> others = (dset1, dset2, dset3)
>>> for dset in others:
>>>     [a.pop('segmentation', None) for a in dset.index.anns.values()]
>>>     [a.pop('keypoints', None) for a in dset.index.anns.values()]
>>> cls = self = kwcoco.CocoDataset
>>> merged = cls.union(*others, disjoint_tracks=1)
>>> print('dset1.anns = {}'.format(ub.repr2(dset1.anns, nl=1)))
>>> print('dset2.anns = {}'.format(ub.repr2(dset2.anns, nl=1)))
>>> print('dset3.anns = {}'.format(ub.repr2(dset3.anns, nl=1)))
>>> print('merged.anns = {}'.format(ub.repr2(merged.anns, nl=1)))
```

Example

```
>>> import kwcoco
>>> # Test empty union
>>> empty_union = kwcoco.CocoDataset.union()
>>> assert len(empty_union.index.imgs) == 0
```

Todo:

- [ ] are supercategories broken?
- [ ] reuse image ids where possible
- [ ] reuse annotation / category ids where possible
- [X] handle case where no inputs are given

- [x] disambiguate track-ids
- [x] disambiguate video-ids

---

subset(*self*, *gids*, *copy=False*, *autobuild=True*)

Return a subset of the larger coco dataset by specifying which images to port. All annotations in those images will be taken.

Parameters

- **gids** (*List[int]*) – image-ids to copy into a new dataset
- **copy** (*bool, default=False*) – if True, makes a deep copy of all nested attributes, otherwise makes a shallow copy.
- **autobuild** (*bool, default=True*) – if True will automatically build the fast lookup index.

Example

```
>>> self = CocoDataset.demo()
>>> gids = [1, 3]
>>> sub_dset = self.subset(gids)
>>> assert len(self.index.gid_to_aids) == 3
>>> assert len(sub_dset.gid_to_aids) == 2
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> gids = [1, 2]
>>> sub_dset = self.subset(gids, copy=True)
>>> assert len(sub_dset.index.videos) == 1
>>> assert len(self.index.videos) == 2
```

Example

```
>>> self = CocoDataset.demo()
>>> sub1 = self.subset([1])
>>> sub2 = self.subset([2])
>>> sub3 = self.subset([3])
>>> others = [sub1, sub2, sub3]
>>> rejoined = CocoDataset.union(*others)
>>> assert len(sub1.anns) == 9
>>> assert len(sub2.anns) == 2
>>> assert len(sub3.anns) == 0
>>> assert rejoined.basic_stats() == self.basic_stats()
```

view_sql(*self*, *force_rewrite=False*, *memory=False*)

Create a cached SQL interface to this dataset suitable for large scale multiprocessing use cases.

Parameters

- **force_rewrite** (*bool, default=False*) – if True, forces an update to any existing cache file on disk
- **memory** (*bool, default=False*) – if True, the database is constructed in memory.

Note: This view cache is experimental and currently depends on the timestamp of the file pointed to by `self.fpath`. In other words dont use this on in-memory datasets.

kwcoco.coco_dataset.**_delitems**(*items*, *remove_idxs*, *thresh=750*)

Parameters

- **items** (*List*) – list which will be modified
- **remove_idxs** (*List[int]*) – integers to remove (MUST BE UNIQUE)

kwcoco.coco_dataset.**demo_coco_data**()

Simple data for testing.

This contains several non-standard fields, which help ensure robustness of functions tested with this data. For more compliant demodata see the `kwcoco.demodata` submodule

Example

```
>>> # xdoctest: +REQUIRES(--show)
>>> from kwcoco.coco_dataset import demo_coco_data, CocoDataset
>>> dataset = demo_coco_data()
>>> self = CocoDataset(dataset, tag='demo')
>>> import kwplot
>>> kwplot.autompl()
>>> self.show_image(gid=1)
>>> kwplot.show_if_requested()
```

2.3.6 kwcoco.coco_evaluator

Evaluates a predicted coco dataset against a truth coco dataset.

The components in this module work programatically or as a command line script.

Todo:

- **[ ] does evaluate return one result or multiple results** based on different configurations?
- [ ] max_dets - TODO: in original pycocoutils but not here
- **[ ] How do we note what iou_thresh and area-range were in** the result plots?

Module Contents

Classes

CocoEvalConfig	Evaluate and score predicted versus truth detections / classifications in a COCO dataset
CocoEvaluator	Abstracts the evaluation process to execute on two coco datasets.
CocoResults	**Example**
CocoSingleResult	Container class to store, draw, summarize, and serialize results from
CocoEvalCLIConfig	Evaluate detection metrics using a predicted and truth coco file.

Functions

dmet_area_weights(dmet, orig_weights, cfsn_vecs, area_ranges, coco_eval, use_area_attr=False)	Hacky function to compute confusion vector ignore weights for different
_load_dets(pred_fpaths, workers=0)	**Example**
_load_dets_worker(single_pred_fpath, with_coco=True)	Ignore:
main(cmdline=True, **kw)	TODO: should live in kwcoco.cli.coco_eval

Attributes

COCO_SAMPLER_CLS

kwcoco.coco_evaluator.**COCO_SAMPLER_CLS**

class kwcoco.coco_evaluator.**CocoEvalConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Evaluate and score predicted versus truth detections / classifications in a COCO dataset

default

normalize(*self*)

overloadable function called after each load

class kwcoco.coco_evaluator.**CocoEvaluator**(*coco_eval*, *config*)

Bases: object

Abstracts the evaluation process to execute on two coco datasets.

This can be run as a standalone script where the user specifies the paths to the true and predited dataset explicitly, or this can be used by a higher level script that produces the predictions and then sends them to this evaluator.

Example

```
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> from kwcoco.demo.perterb import perterb_coco
>>> import kwcoco
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>>     'with_probs': True,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'classes_of_interest': [],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
```

Config

log(*coco_eval*, *msg*, *level='INFO'*)

_init(*coco_eval*)

Performs initial coercion from given inputs into dictionaries of kwimage.Detection objects and attempts to ensure comparable category and image ids.

_ensure_init(*coco_eval*)

classmethod _rectify_classes(*coco_eval*, *true_classes*, *pred_classes*)

classmethod _coerce_dets(*CocoEvaluator*, *dataset*, *verbose=0*, *workers=0*)

Coerce the input to a mapping from image-id to kwimage.Detection

Also capture a CocoDataset if possible.

Returns gid_to_det: mapping from gid to dets extra: any extra information we gathered via coercion

Return type Tuple[Dict[int, Detections], Dict]

Example

```
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.demo('shapes8')
>>> gid_to_det, extras = CocoEvaluator._coerce_dets(coco_dset)
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.demo('shapes8').view_sql()
>>> gid_to_det, extras = CocoEvaluator._coerce_dets(coco_dset)
```

_build_dmet(*coco_eval*)

Builds the detection metrics object

Returns

DetectionMetrics - object that can perform assignment and build confusion vectors.

Ignore: dmet = coco_eval._build_dmet()

evaluate(*coco_eval*)

Executes the main evaluation logic. Performs assignments between detections to make DetectionMetrics object, then creates per-item and ovr confusion vectors, and performs various threshold-vs-confusion analyses.

Returns

container storing (and capable of drawing / serializing) results

Return type *CocoResults*

CommandLine: xdoctest -m kwcoco.coco_evaluator CocoEvaluator.evaluate –vd

Example

```
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> true_dset = kwcoco.CocoDataset.demo('shapes128')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>>     'with_probs': True,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> print('true_dset = {!r}'.format(true_dset))
>>> print('pred_dset = {!r}'.format(pred_dset))
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'area_range': ['all', 'small'],
>>>     'iou_thresh': [0.3, 0.95],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
```

(continues on next page)

(continued from previous page)

```
>>> # Now we can draw / serialize the results as we please
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/test_out_dpath')
>>> results.dump(join(dpath, 'metrics.json'), indent='    ')
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> # xdoctest: +REQUIRES(--slow)
>>> results.dump_figures(dpath)
>>> # xdoctest: +REQUIRES(--vd)
>>> if ub.argflag('--vd') or 1:
>>>     import xdev
>>>     xdev.view_directory(dpath)
```

kwcoco.coco_evaluator.**dmet_area_weights**(*dmet*, *orig_weights*, *cfsn_vecs*, *area_ranges*, *coco_eval*, *use_area_attr=False*)

Hacky function to compute confusion vector ignore weights for different area thresholds. Needs to be slightly refactored.

class kwcoco.coco_evaluator.**CocoResults**(*results*, *resdata=None*)

Bases: ubelt.NiceRepr, *kwcoco.metrics.util.DictProxy*

Example

```
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> true_dset = kwcoco.CocoDataset.demo('shapes2')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.5,
>>>     'n_fp': (0, 10),
>>>     'n_fn': (0, 10),
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> print('true_dset = {!r}'.format(true_dset))
>>> print('pred_dset = {!r}'.format(pred_dset))
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'area_range': ['small'],
>>>     'iou_thresh': [0.3],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
>>> # Now we can draw / serialize the results as we please
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/test_out_dpath')
>>> #
>>> # test deserialization works
>>> state = results.__json__()
>>> self2 = CocoResults.from_json(state)
>>> #
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> results.dump_figures(dpath)
>>> results.dump(join(dpath, 'metrics.json'), indent='    ')
```

dump_figures(*results*, *out_dpath*, *expt_title=None*)

__json__(*results*)

classmethod from_json(*cls*, *state*)

dump(*result*, *file*, *indent=' '*)
Serialize to json file

class kwcoco.coco_evaluator.**CocoSingleResult**(*result*, *nocls_measures*, *ovr_measures*, *cfsn_vecs*, *meta=None*)

Bases: ubelt.NiceRepr

Container class to store, draw, summarize, and serialize results from CocoEvaluator.

Ignore:

```
>>> from kwcoco.coco_evaluator import *  # NOQA
>>> from kwcoco.coco_evaluator import CocoEvaluator
>>> import kwcoco
>>> true_dset = kwcoco.CocoDataset.demo('shapes8')
>>> from kwcoco.demo.perterb import perterb_coco
>>> kwargs = {
>>>     'box_noise': 0.2,
>>>     'n_fp': (0, 3),
>>>     'n_fn': (0, 3),
>>>     'with_probs': False,
>>> }
>>> pred_dset = perterb_coco(true_dset, **kwargs)
>>> print('true_dset = {!r}'.format(true_dset))
>>> print('pred_dset = {!r}'.format(pred_dset))
>>> config = {
>>>     'true_dataset': true_dset,
>>>     'pred_dataset': pred_dset,
>>>     'area_range': [(0, 32 ** 2), (32 ** 2, 96 ** 2)],
>>>     'iou_thresh': [0.3, 0.5, 0.95],
>>> }
>>> coco_eval = CocoEvaluator(config)
>>> results = coco_eval.evaluate()
>>> result = ub.peek(results.values())
>>> state = result.__json__()
>>> print('state = {}'.format(ub.repr2(state, nl=-1)))
>>> recon = CocoSingleResult.from_json(state)
>>> state = recon.__json__()
>>> print('state = {}'.format(ub.repr2(state, nl=-1)))
```

__nice__(*result*)

classmethod from_json(*cls*, *state*)

__json__(*result*)

dump(*result*, *file*, *indent=' '*)
Serialize to json file

dump_figures(*result*, *out_dpath*, *expt_title=None*)

kwcoco.coco_evaluator.**_load_dets**(*pred_fpaths*, *workers=0*)

Example

```
>>> from kwcoco.coco_evaluator import _load_dets, _load_dets_worker
>>> import ubelt as ub
>>> import kwcoco
>>> from os.path import join
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/load_dets')
>>> N = 4
>>> pred_fpaths = []
>>> for i in range(1, N + 1):
>>>     dset = kwcoco.CocoDataset.demo('shapes{}'.format(i))
>>>     dset.fpath = join(dpath, 'shapes_{}.mscoco.json'.format(i))
>>>     dset.dump(dset.fpath)
>>>     pred_fpaths.append(dset.fpath)
>>> dets, coco_dset = _load_dets(pred_fpaths)
>>> print('dets = {!r}'.format(dets))
>>> print('coco_dset = {!r}'.format(coco_dset))
```

kwcoco.coco_evaluator.**_load_dets_worker**(*single_pred_fpath*, *with_coco=True*)

Ignore:

```
>>> from kwcoco.coco_evaluator import _load_dets, _load_dets_worker
>>> import ubelt as ub
>>> import kwcoco
>>> from os.path import join
>>> dpath = ub.ensure_app_cache_dir('kwcoco/tests/load_dets')
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> dset.fpath = join(dpath, 'shapes8.mscoco.json')
>>> dset.dump(dset.fpath)
>>> single_pred_fpath = dset.fpath
>>> dets, coco = _load_dets_worker(single_pred_fpath)
>>> print('dets = {!r}'.format(dets))
>>> print('coco = {!r}'.format(coco))
```

class kwcoco.coco_evaluator.**CocoEvalCLIConfig**(*data=None*, *default=None*, *cmdline=False*)

Bases: scriptconfig.Config

Evaluate detection metrics using a predicted and truth coco file.

default

kwcoco.coco_evaluator.**main**(*cmdline=True*, ***kw*)

TODO: should live in kwcoco.cli.coco_eval

CommandLine:

Generate test data xdoctest -m kwcoco.cli.coco_eval CocoEvalCLI.main

kwcoco eval –true_dataset=$HOME/.cache/kwcoco/tests/eval/true.mscoco.json –pred_dataset=$HOME/.cache/kwcoco/tests/eval/pred.mscoco.json –out_dpath=$HOME/.cache/kwcoco/tests/eval/out –force_pycocoutils=False –area_range=all,0-4096,4096-inf

nautilus $HOME/.cache/kwcoco/tests/eval/out

2.3.7 kwcoco.coco_image

Module Contents

Classes

CocoImage	An object-oriented representation of a coco image.

class kwcoco.coco_image.**CocoImage**(*img*, *dset=None*)

Bases: ubelt.NiceRepr

An object-oriented representation of a coco image.

It provides helper methods that are specific to a single image.

This operates directly on a single coco image dictionary, but it can optionally be connected to a parent dataset, which allows it to use CocoDataset methods to query about relationships and resolve pointers.

This is different than the Images class in coco_object1d, which is just a vectorized interface to multiple objects.

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> self = dset._coco_image(1)
```

```
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> self = dset._coco_image(1)
```

__nice__(*self*)

__getitem__(*self*, *key*)

get(*self*, *key*, *default=ub.NoParam*)

Duck type some of the dict interface

property dsize(*self*)

_iter_asset_objs(*self*)

Iterate through base + auxiliary dicts that have file paths

2.3.8 kwcoco.coco_objects1d

Vectorized ORM-like objects used in conjunction with coco_dataset

Module Contents

Classes

ObjectList1D	Vectorized access to lists of dictionary objects
ObjectGroups	An object for holding a groups of *ObjectList1D* objects
Categories	Vectorized access to category attributes
Videos	Vectorized access to video attributes
Images	Vectorized access to image attributes
Annots	Vectorized access to annotation attributes
AnnotGroups	An object for holding a groups of *ObjectList1D* objects
ImageGroups	An object for holding a groups of *ObjectList1D* objects

class kwcoco.coco_objects1d.**ObjectList1D**(*ids*, *dset*, *key*)

Bases: ubelt.NiceRepr

Vectorized access to lists of dictionary objects

Lightweight reference to a set of object (e.g. annotations, images) that allows for convenient property access.

Parameters

- **ids** (*List[int]*) – list of ids
- **dset** (*CocoDataset*) – parent dataset
- **key** (*str*) – main object name (e.g. 'images', 'annotations')

Types: ObjT = Ann | Img | Cat # can be one of these types ObjectList1D gives us access to a List[ObjT]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> # Both annots and images are object lists
>>> self = dset.annots()
>>> self = dset.images()
>>> # can call with a list of ids or not, for everything
>>> self = dset.annots([1, 2, 11])
>>> self = dset.images([1, 2, 3])
>>> self.lookup('id')
>>> self.lookup(['id'])
```

__nice__(*self*)

__iter__(*self*)

__len__(*self*)

property _id_to_obj(*self*)

property objs(*self*)

Returns all object dictionaries

Return type List

take(*self*, *idxs*)
Take a subset by index

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().annots()
>>> assert len(self.take([0, 2, 3])) == 3
```

compress(*self*, *flags*)
Take a subset by flags

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> assert len(self.compress([True, False, True])) == 2
```

peek(*self*)
Return the first object dictionary

lookup(*self*, *key*, *default=ub.NoParam*, *keepid=False*)
Lookup a list of object attributes

Parameters

- **key** (*str | Iterable*) – name of the property you want to lookup can also be a list of names, in which case we return a dict
- **default** – if specified, uses this value if it doesn't exist in an ObjT.
- **keepid** – if True, return a mapping from ids to the property

Returns a list of whatever type the object is Dict[str, ObjT]

Return type List[ObjT]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.annots()
>>> self.lookup('id')
>>> key = ['id']
>>> default = None
>>> self.lookup(key=['id', 'image_id'])
>>> self.lookup(key=['id', 'image_id'])
>>> self.lookup(key='foo', default=None, keepid=True)
>>> self.lookup(key=['foo'], default=None, keepid=True)
>>> self.lookup(key=['id', 'image_id'], keepid=True)
```

get(*self*, *key*, *default=ub.NoParam*, *keepid=False*)
Lookup a list of object attributes

Parameters

- **key** (*str*) – name of the property you want to lookup
- **default** – if specified, uses this value if it doesn't exist in an ObjT.
- **keepid** – if True, return a mapping from ids to the property

Returns a list of whatever type the object is Dict[str, ObjT]

Return type List[ObjT]

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.annots()
>>> self.get('id')
>>> self.get(key='foo', default=None, keepid=True)
```

set(*self*, *key*, *values*)
Assign a value to each annotation

Parameters

- **key** (*str*) – the annotation property to modify
- **values** (*Iterable | scalar*) – an iterable of values to set for each annot in the dataset. If the item is not iterable, it is assigned to all objects.

Example

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> self = dset.annots()
>>> self.set('my-key1', 'my-scalar-value')
>>> self.set('my-key2', np.random.rand(len(self)))
>>> print('dset.imgs = {}'.format(ub.repr2(dset.imgs, nl=1)))
>>> self.get('my-key2')
```

_set(*self*, *key*, *values*)
faster less safe version of set

_lookup(*self*, *key*, *default=ub.NoParam*)

Benchmark:

```
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes256')
>>> self = annots = dset.annots()
```

```
>>> import timerit
>>> ti = timerit.Timerit(100, bestof=10, verbose=2)
```

for timer in ti.reset('lookup'):

with timer: self.lookup('image_id')

for timer in ti.reset('_lookup'):

with timer: self._lookup('image_id')

for timer in ti.reset('image_id'):

with timer: self.image_id

for timer in ti.reset('raw1'):

with timer: key = 'image_id' [self._dset.anns[_id][key] for _id in self._ids]

for timer in ti.reset('raw2'):

with timer: anns = self._dset.anns key = 'image_id' [anns[_id][key] for _id in self._ids]

for timer in ti.reset('lut-gen'):

with timer: _lut = self._obj_lut objs = (_lut[_id] for _id in self._ids) [obj[key] for obj in objs]

for timer in ti.reset('lut-gen-single'):

with timer: _lut = self._obj_lut [_lut[_id][key] for _id in self._ids]

class kwcoco.coco_objects1d.**ObjectGroups**(*groups*, *dset*)

Bases: ubelt.NiceRepr

An object for holding a groups of *ObjectList1D* objects

_lookup(*self*, *key*)

__getitem__(*self*, *index*)

lookup(*self*, *key*, *default=ub.NoParam*)

__nice__(*self*)

class kwcoco.coco_objects1d.**Categories**(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to category attributes

Example

```
>>> from kwcoco.coco_objects1d import Categories  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo()
>>> ids = list(dset.cats.keys())
>>> self = Categories(ids, dset)
>>> print('self.name = {!r}'.format(self.name))
>>> print('self.supercategory = {!r}'.format(self.supercategory))
```

property cids(*self*)

property name(*self*)

property supercategory(*self*)

class kwcoco.coco_objects1d.**Videos**(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to video attributes

Example

```
>>> from kwcoco.coco_objects1d import Videos  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes5')
>>> ids = list(dset.index.videos.keys())
>>> self = Videos(ids, dset)
>>> print('self = {!r}'.format(self))
```

class kwcoco.coco_objects1d.**Images**(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to image attributes

SeeAlso: kwcoco.CocoDataset.images()

__getitem__(*self*, *index*)

property gids(*self*)

property gname(*self*)

property gpath(*self*)

property width(*self*)

property height(*self*)

property size(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> self._dset._ensure_imgsize()
>>> print(self.size)
[(512, 512), (300, 250), (256, 256)]
```

property area(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> self._dset._ensure_imgsize()
>>> print(self.area)
[262144, 75000, 65536]
```

property n_annots(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> print(ub.repr2(self.n_annots, nl=0))
[9, 2, 0]
```

property aids(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> print(ub.repr2(list(map(list, self.aids)), nl=0))
[[1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 11], []]
```

property annots(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().images()
>>> print(self.annots)
<AnnotGroups(n=3, m=3.7, s=3.9)>
```

class kwcoco.coco_objects1d.**Annots**(*ids*, *dset*)

Bases: *ObjectList1D*

Vectorized access to annotation attributes

property aids(*self*)

The annotation ids of this column of annotations

property images(*self*)

Get the column of images

Returns Images

property image_id(*self*)

property category_id(*self*)

property gids(*self*)
Get the column of image-ids

Returns list of image ids

Return type List[int]

property cids(*self*)
Get the column of category-ids

Returns List[int]

property cnames(*self*)
Get the column of category names

Returns List[int]

property detections(*self*)
Get the kwimage-style detection objects

Returns kwimage.Detections

Example

```
>>> # xdoctest: +REQUIRES(module:kwimage)
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('shapes32').annots([1, 2, 11])
>>> dets = self.detections
>>> print('dets.data = {!r}'.format(dets.data))
>>> print('dets.meta = {!r}'.format(dets.meta))
```

property boxes(*self*)
Get the column of kwimage-style bounding boxes

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().annots([1, 2, 11])
>>> print(self.boxes)
<Boxes(xywh,
    array([[ 10,  10, 360, 490],
           [350,   5, 130, 290],
           [124,  96,  45,  18]]))>
```

property xywh(*self*)
Returns raw boxes

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo().annots([1, 2, 11])
>>> print(self.xywh)
```

class kwcoco.coco_objects1d.**AnnotGroups**(*groups*, *dset*)

Bases: *ObjectGroups*

An object for holding a groups of *ObjectList1D* objects

property cids(*self*)

class kwcoco.coco_objects1d.**ImageGroups**(*groups*, *dset*)

Bases: *ObjectGroups*

An object for holding a groups of *ObjectList1D* objects

2.3.9 kwcoco.coco_schema

CommandLine: python -m kwcoco.coco_schema xdoctest -m kwcoco.coco_schema __doc__

Example

```
>>> import kwcoco
>>> from kwcoco.coco_schema import COCO_SCHEMA
>>> import jsonschema
>>> dset = kwcoco.CocoDataset.demo('shapes1')
>>> # print('dset.dataset = {}'.format(ub.repr2(dset.dataset, nl=2)))
>>> COCO_SCHEMA.validate(dset.dataset)
```

```
>>> try:
>>>     jsonschema.validate(dset.dataset, schema=COCO_SCHEMA)
>>> except jsonschema.exceptions.ValidationError as ex:
>>>     vali_ex = ex
>>>     print('ex = {!r}'.format(ex))
>>>     raise
>>> except jsonschema.exceptions.SchemaError as ex:
>>>     print('ex = {!r}'.format(ex))
>>>     schema_ex = ex
>>>     print('schema_ex.instance = {}'.format(ub.repr2(schema_ex.instance, nl=-1)))
>>>     raise
```

```
>>> # Test the multispectral image defintino
>>> import copy
>>> dataset = dset.copy().dataset
>>> img = dataset['images'][0]
>>> img.pop('file_name')
>>> import pytest
>>> with pytest.raises(jsonschema.ValidationError):
>>>     COCO_SCHEMA.validate(dataset)
>>> import pytest
```

(continues on next page)

(continued from previous page)

```
>>> img['auxiliary'] = [{'file_name': 'foobar'}]
>>> with pytest.raises(jsonschema.ValidationError):
>>>     COCO_SCHEMA.validate(dataset)
>>> img['name'] = 'aux-only images must have a name'
>>> COCO_SCHEMA.validate(dataset)
```

Module Contents

Functions

deprecated(*args)

TUPLE(*args, **kw)

Attributes

elem

ALLOF

ANY

ANYOF

ARRAY

BOOLEAN

INTEGER

NOT

NULL

NUMBER

OBJECT

ONEOF

STRING

UUID

PATH

continues on next page

Table 86 – continued from previous page

KWCOCO_KEYPOINT
KWCOCO_POLYGON
ORIG_COCO_KEYPOINTS
KWCOCO_KEYPOINTS
KEYPOINTS
ORIG_COCO_POLYGON
POLYGON
RUN_LENGTH_ENCODING
BBOX
SEGMENTATION
CATEGORY
KEYPOINT_CATEGORY
VIDEO
CHANNELS
IMAGE
ANNOTATION
COCO_SCHEMA

kwcoco.coco_schema.**deprecated**(**args*)

kwcoco.coco_schema.**TUPLE**(**args*, ***kw*)

kwcoco.coco_schema.**elem**

kwcoco.coco_schema.**ALLOF**

kwcoco.coco_schema.**ANY**

kwcoco.coco_schema.**ANYOF**

kwcoco.coco_schema.**ARRAY**

kwcoco.coco_schema.**BOOLEAN**

kwcoco.coco_schema.**INTEGER**

kwcoco.coco_schema.**NOT**

kwcoco.coco_schema.**NULL**

kwcoco.coco_schema.**NUMBER**

kwcoco.coco_schema.**OBJECT**

kwcoco.coco_schema.**ONEOF**

kwcoco.coco_schema.**STRING**

kwcoco.coco_schema.**UUID**

kwcoco.coco_schema.**PATH**

kwcoco.coco_schema.**KWCOCO_KEYPOINT**

kwcoco.coco_schema.**KWCOCO_POLYGON**

kwcoco.coco_schema.**ORIG_COCO_KEYPOINTS**

kwcoco.coco_schema.**KWCOCO_KEYPOINTS**

kwcoco.coco_schema.**KEYPOINTS**

kwcoco.coco_schema.**ORIG_COCO_POLYGON**

kwcoco.coco_schema.**POLYGON**

kwcoco.coco_schema.**RUN_LENGTH_ENCODING**

kwcoco.coco_schema.**BBOX**

kwcoco.coco_schema.**SEGMENTATION**

kwcoco.coco_schema.**CATEGORY**

kwcoco.coco_schema.**KEYPOINT_CATEGORY**

kwcoco.coco_schema.**VIDEO**

kwcoco.coco_schema.**CHANNELS**

kwcoco.coco_schema.**IMAGE**

kwcoco.coco_schema.**ANNOTATION**

kwcoco.coco_schema.**COCO_SCHEMA**

2.3.10 kwcoco.coco_sql_dataset

---

speed difference so we can take advantage of alchemy's expressiveness?:

Finally got a baseline implementation of an SQLite backend for COCO datasets. This mostly plugs into my existing tools (as long as only read operations are used; haven't impelmented writing yet) by duck-typing the dict API.

This solves the issue of forking and then accessing nested dictionaries in the JSON-style COCO objects. (When you access the dictionary Python will increment a reference count which triggers copy-on-write for whatever memory page that data happened to live in. Non-contiguous access had the effect of excessive memory copies).

For "medium sized" datasets its quite a bit slower. Running through a torch DataLoader with 4 workers for 10,000 images executes at a rate of 100Hz but takes 850MB of RAM. Using the duck-typed SQL backend only uses 500MB (which includes the cost of caching), but runs at 45Hz (which includes the benefit of caching).

However, once I scale up to 100,000 images I start seeing benefits. The in-memory dictionary interface chugs at 1.05HZ, and is taking more than 4GB of memory at the time I killed the process (eta was over an hour). The SQL backend ran at 45Hz and took about 3 minutes and used about 2.45GB of memory.

Without a cache, SQL runs at 30HZ and takes 400MB for 10,000 images, and for 100,000 images it gets 30Hz with 1.1GB. There is also a much larger startup time. I'm not exactly sure what it is yet, but its probably some preprocessing I'm doing.

Using a LRU cache we get 45Hz and 1.05GB of memory, so that's a clear win. We do need to be sure to disable the cache if we ever implement write mode.

I'd like to be a bit faster on the medium sized datasets (I'd really like to avoid caching rows, which is why the speed is currently semi-reasonable), but I don't think I can do any better than this because single-row lookup time is *O(log(N))* for sqlite, whereas its *O(1)* for dictionaries. (I wish sqlite had an option to create a hash-table index for a table, but I dont think it does). I optimized as many of the dictionary operations as possible (for instance, iterating through keys, values, and items should be O(N) instead of *O(N log(N))*), but the majority of the runtime cost is in the single-row lookup time.

There are a few questions I still have if anyone has insight:

- Say I want to select a subset of K rows from a table with N entries, and I have a list of all of the rowids that I want. Is there any way to do this better than *O(K log(N))*? I tried using a *SELECT col FROM table WHERE id IN (?, ?, ?, ?, ...)* filling in enough *?* as there are rows in my subset. I'm not sure what the complexity of using a query like this is. I'm not sure what the *IN* implementation looks like. Can this be done more efficiently by with a temporary table and a *JOIN*?
- There really is no way to do *O(1)* row lookup in sqlite right? Is there a way in PostgreSQL or some other backend sqlalchemy supports?

I found that PostgreSQL does support hash indexes: https://www.postgresql.org/docs/13/indexes-types.html I'm really not interested in setting up a global service though . I also found a 10-year old thread with a hash-index feature request for SQLite, which I unabashedly resurrected http://sqlite.1065341.n5.nabble.com/Feature-request-hash-index-td23367.html

Module Contents

Classes

Category	
KeypointCategory	
Video	
Image	
Annotation	
SqlListProxy	A view of an SQL table that behaves like a Python list
SqlDictProxy	Duck-types an SQL table as a dictionary of dictionaries.
SqlIdGroupDictProxy	Similar to *SqlDictProxy*, but maps ids to groups of other ids.
CocoSqlIndex	Simulates the dictionary provided by *kwcoco.coco_dataset.CocoIndex*
CocoSqlDatabase	Provides an API nearly identical to `kwcoco.CocoDatabase`, but uses

Functions

orm_to_dict(obj)	
_orm_yielder(query, size=300)	TODO: figure out the best way to yield, in batches or otherwise
_raw_yielder(result, size=300)	TODO: figure out the best way to yield, in batches or otherwise
_new_proxy_cache()	By returning None, we wont use item caching
_handle_sql_uri(uri)	Temporary function to deal with URI. Modern tools seem to use RFC 3968
cached_sql_coco_view(dct_db_fpath=None, sql_db_fpath=None, dset=None, force_rewrite=False)	Attempts to load a cached SQL-View dataset, only loading and converting the
ensure_sql_coco_view(dset, db_fpath=None, force_rewrite=False)	Create a cached on-disk SQL view of an on-disk COCO dataset.
demo(num=10)	
assert_dsets_allclose(dset1, dset2, tag1='dset1', tag2='dset2')	
_benchmark_dset_readtime(dset, tag='?')	Helper for understanding the time differences between backends
_benchmark_dict_proxy_ops(proxy)	Get insight on the efficiency of operations
devcheck()	Scratch work for things that should eventually become unit or doc tests

Attributes

CocoBase	
ALCHEMY_MODE_DEFAULT	
TBLNAME_TO_CLASS	
tblname	

kwcoco.coco_sql_dataset.**CocoBase**

class kwcoco.coco_sql_dataset.**Category**

Bases: *CocoBase*

__tablename__ = categories

id

name

alias

supercategory

extra

class kwcoco.coco_sql_dataset.KeypointCategory
Bases: *CocoBase*

__tablename__ = keypoint_categories

id

name

alias

supercategory

reflection_id

extra

class kwcoco.coco_sql_dataset.Video
Bases: *CocoBase*

__tablename__ = videos

id

name

caption

width

height

extra

class kwcoco.coco_sql_dataset.Image
Bases: *CocoBase*

__tablename__ = images

id

name

file_name

width

height

video_id

timestamp

frame_index

channels

auxiliary

extra

class kwcoco.coco_sql_dataset.Annotation
Bases: *CocoBase*

__tablename__ = annotations

id

image_id

category_id

track_id

segmentation

keypoints

bbox

_bbox_x

_bbox_y

_bbox_w

_bbox_h

weight

score

weight

prob

iscrowd

caption

extra

kwcoco.coco_sql_dataset.**ALCHEMY_MODE_DEFAULT = 0**

kwcoco.coco_sql_dataset.**TBLNAME_TO_CLASS**

kwcoco.coco_sql_dataset.**tblname**

kwcoco.coco_sql_dataset.**orm_to_dict**(*obj*)

kwcoco.coco_sql_dataset.**_orm_yielder**(*query*, *size=300*)
TODO: figure out the best way to yield, in batches or otherwise

kwcoco.coco_sql_dataset.**_raw_yielder**(*result*, *size=300*)
TODO: figure out the best way to yield, in batches or otherwise

kwcoco.coco_sql_dataset.**_new_proxy_cache**()
By returning None, we wont use item caching

class kwcoco.coco_sql_dataset.**SqlListProxy**(*proxy*, *session*, *cls*)
Bases: ubelt.NiceRepr

A view of an SQL table that behaves like a Python list

__len__(*proxy*)

__nice__(*proxy*)

__iter__(*proxy*)

__getitem__(*proxy*, *index*)

__contains__(*proxy*, *item*)

__setitem__(*proxy*, *index*, *value*)

__delitem__(*proxy*, *index*)

class kwcoco.coco_sql_dataset.**SqlDictProxy**(*proxy*, *session*, *cls*, *keyattr=None*, *ignore_null=False*)

Bases: *kwcoco.util.dict_like.DictLike*

Duck-types an SQL table as a dictionary of dictionaries.

The key is specified by an indexed column (by default it is the *id* column). The values are dictionaries containing all data for that row.

Notes

With SQLite indexes are B-Trees so lookup is O(log(N)) and not O(1) as will regular dictionaries. Iteration should still be O(N), but databases have much more overhead than Python dictionaries.

Parameters

- **session** (*Session*) – the sqlalchemy session
- **cls** (*Type*) – the declarative sqlalchemy table class
- **keyattr** – the indexed column to use as the keys
- **ignore_null** (*bool*) – if True, ignores any keys set to NULL, otherwise NULL keys are allowed.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import pytest
>>> sql_dset, dct_dset = demo(num=10)
>>> proxy = sql_dset.index.anns
```

```
>>> keys = list(proxy.keys())
>>> values = list(proxy.values())
>>> items = list(proxy.items())
>>> item_keys = [t[0] for t in items]
>>> item_vals = [t[1] for t in items]
>>> lut_vals = [proxy[key] for key in keys]
>>> assert item_vals == lut_vals == values
>>> assert item_keys == keys
>>> assert len(proxy) == len(keys)
```

```
>>> goodkey1 = keys[1]
>>> badkey1 = -100000000000
>>> badkey2 = 'foobarbazbiz'
>>> badkey3 = object()
>>> assert goodkey1 in proxy
>>> assert badkey1 not in proxy
>>> assert badkey2 not in proxy
>>> assert badkey3 not in proxy
>>> with pytest.raises(KeyError):
>>>     proxy[badkey1]
>>> with pytest.raises(KeyError):
>>>     proxy[badkey2]
```

(continues on next page)

(continued from previous page)

```
>>> with pytest.raises(KeyError):
>>>     proxy[badkey3]
```

```
>>> # xdoctest: +SKIP
>>> from kwcoco.coco_sql_dataset import _benchmark_dict_proxy_ops
>>> ti = _benchmark_dict_proxy_ops(proxy)
>>> print('ti.measures = {}'.format(ub.repr2(ti.measures, nl=2, align=':',␣
↪precision=6)))
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import kwcoco
>>> # Test the variant of the SqlDictProxy where we ignore None keys
>>> # This is the case for name_to_img and file_name_to_img
>>> dct_dset = kwcoco.CocoDataset.demo('shapes1')
>>> dct_dset.add_image(name='no_file_image1')
>>> dct_dset.add_image(name='no_file_image2')
>>> dct_dset.add_image(name='no_file_image3')
>>> sql_dset = dct_dset.view_sql(memory=True)
>>> assert len(dct_dset.index.imgs) == 4
>>> assert len(dct_dset.index.file_name_to_img) == 1
>>> assert len(dct_dset.index.name_to_img) == 3
>>> assert len(sql_dset.index.imgs) == 4
>>> assert len(sql_dset.index.file_name_to_img) == 1
>>> assert len(sql_dset.index.name_to_img) == 3
```

```
>>> proxy = sql_dset.index.file_name_to_img
>>> assert len(list(proxy.keys())) == 1
>>> assert len(list(proxy.values())) == 1
```

```
>>> proxy = sql_dset.index.name_to_img
>>> assert len(list(proxy.keys())) == 3
>>> assert len(list(proxy.values())) == 3
```

```
>>> proxy = sql_dset.index.imgs
>>> assert len(list(proxy.keys())) == 4
>>> assert len(list(proxy.values())) == 4
```

items

values

__len__(*proxy*)

__nice__(*proxy*)

__contains__(*proxy*, *key*)

__getitem__(*proxy*, *key*)

keys(*proxy*)

itervalues(*proxy*)

iteritems(*proxy*)

class kwcoco.coco_sql_dataset.**SqlIdGroupDictProxy**(*proxy*, *session*, *valattr*, *keyattr*, *parent_keyattr*, *group_order_attr=None*)

Bases: *kwcoco.util.dict_like.DictLike*

Similar to *SqlDictProxy*, but maps ids to groups of other ids.

Simulates a dictionary that maps ids of a parent table to all ids of another table corresponding to rows where a specific column has that parent id.

The items in the group can be sorted by the `group_order_attr` if specified.

For example, imagine two tables: images with one column (id) and annotations with two columns (id, image_id). This class can help provide a mpaping from each *image.id* to a *Set[annotation.id]* where those annotation rows have *annotation.image_id = image.id*.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> sql_dset, dct_dset = demo(num=10)
>>> proxy = sql_dset.index.gid_to_aids
```

```
>>> keys = list(proxy.keys())
>>> values = list(proxy.values())
>>> items = list(proxy.items())
>>> item_keys = [t[0] for t in items]
>>> item_vals = [t[1] for t in items]
>>> lut_vals = [proxy[key] for key in keys]
>>> assert item_vals == lut_vals == values
>>> assert item_keys == keys
>>> assert len(proxy) == len(keys)
```

```
>>> # xdoctest: +SKIP
>>> from kwcoco.coco_sql_dataset import _benchmark_dict_proxy_ops
>>> ti = _benchmark_dict_proxy_ops(proxy)
>>> print('ti.measures = {}'.format(ub.repr2(ti.measures, nl=2, align=':',␣
↪precision=6)))
```

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> import kwcoco
>>> # Test the group sorted variant of this by using vidid_to_gids
>>> # where the "gids" must be sorted by the image frame indexes
>>> dct_dset = kwcoco.CocoDataset.demo('vidshapes1')
>>> dct_dset.add_image(name='frame-index-order-demo1', frame_index=-30, video_id=1)
>>> dct_dset.add_image(name='frame-index-order-demo2', frame_index=10, video_id=1)
>>> dct_dset.add_image(name='frame-index-order-demo3', frame_index=3, video_id=1)
```

(continues on next page)

(continued from previous page)

```
>>> dct_dset.add_video(name='empty-video1')
>>> dct_dset.add_video(name='empty-video2')
>>> dct_dset.add_video(name='empty-video3')
>>> sql_dset = dct_dset.view_sql(memory=True)
>>> orig = dct_dset.index.vidid_to_gids
>>> proxy = sql_dset.index.vidid_to_gids
>>> from kwcoco.util.util_json import indexable_allclose
>>> assert indexable_allclose(orig, dict(proxy))
>>> items = list(proxy.iteritems())
>>> vals = list(proxy.itervalues())
>>> keys = list(proxy.iterkeys())
>>> assert len(keys) == len(vals)
>>> assert dict(zip(keys, vals)) == dict(items)
```

__nice__(*self*)

__len__(*proxy*)

__getitem__(*proxy*, *key*)

__contains__(*proxy*, *key*)

keys(*proxy*)

iteritems(*proxy*)

itervalues(*proxy*)

class kwcoco.coco_sql_dataset.**CocoSqlIndex**(*index*)

Bases: object

Simulates the dictionary provided by *kwcoco.coco_dataset.CocoIndex*

build(*index*, *parent*)

kwcoco.coco_sql_dataset.**_handle_sql_uri**(*uri*)

Temporary function to deal with URI. Modern tools seem to use RFC 3968 URIs, but sqlalchemy uses RFC 1738. Attempt to gracefully handle special cases. With a better understanding of the above specs, this function may be able to be written more eloquently.

Ignore: _handle_sql_uri(':memory:') _handle_sql_uri('special:foobar') _handle_sql_uri('sqlite:///:memory:') _handle_sql_uri('/foo/bar') _handle_sql_uri('foo/bar')

class kwcoco.coco_sql_dataset.**CocoSqlDatabase**(*uri=None*, *tag=None*, *img_root=None*)

Bases: *kwcoco.abstract_coco_dataset.AbstractCocoDataset*, *kwcoco.coco_dataset.MixinCocoAccessors*, *kwcoco.coco_dataset.MixinCocoObjects*, *kwcoco.coco_dataset.MixinCocoStats*, *kwcoco.coco_dataset.MixinCocoDraw*, ubelt.NiceRepr

Provides an API nearly identical to kwcoco.CocoDatabase, but uses an SQL backend data store. This makes it robust to copy-on-write memory issues that arise when forking, as discussed in[1].

[1] https://github.com/pytorch/pytorch/issues/13246

Notes

By default constructing an instance of the CocoSqlDatabase does not create a connection to the databse. Use the *connect()* method to open a connection.

References

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> sql_dset, dct_dset = demo()
>>> assert_dsets_allclose(sql_dset, dct_dset)
```

MEMORY_URI = sqlite:///:memory:

__nice__(*self*)

__getstate__(*self*)

Return only the minimal info when pickling this object.

Note: This object IS pickling when the multiprocessing context is "spawn".

This object is NOT pickled when the multiprocessing context is "fork". In this case the user needs to be careful to create new connections in the forked subprocesses.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> sql_dset, dct_dset = demo()
>>> # Test pickling works correctly
>>> import pickle
>>> serialized = pickle.dumps(sql_dset)
>>> assert len(serialized) < 3e4, 'should be very small'
>>> copy = pickle.loads(serialized)
>>> dset1, dset2, tag1, tag2 = sql_dset, copy, 'orig', 'copy'
>>> assert_dsets_allclose(dset1, dset2, tag1, tag2)
>>> # --- other methods of copying ---
>>> rw_copy = CocoSqlDatabase(
>>>     sql_dset.uri, img_root=sql_dset.img_root, tag=sql_dset.tag)
>>> rw_copy.connect()
>>> ro_copy = CocoSqlDatabase(
>>>     sql_dset.uri, img_root=sql_dset.img_root, tag=sql_dset.tag)
>>> ro_copy.connect(readonly=True)
>>> assert_dsets_allclose(dset1, ro_copy, tag1, 'ro-copy')
>>> assert_dsets_allclose(dset1, rw_copy, tag1, 'rw-copy')
```

__setstate__(*self*, *state*)

Reopen new readonly connnections when unpickling the object.

disconnect(*self*)
Drop references to any SQL or cache objects

connect(*self*, *readonly=False*)
Connects this instance to the underlying database.

References

details on read only mode, some of these didnt seem to work https://github.com/sqlalchemy/sqlalchemy/blob/master/lib/sqlalchemy/dialects/sqlite/pysqlite.py#L71 https://github.com/pudo/dataset/issues/136 https://writeonly.wordpress.com/2009/07/16/simple-read-only-sqlalchemy-sessions/

property fpath(*self*)

delete(*self*)

populate_from(*self*, *dset*, *verbose=1*)
Copy the information in a `CocoDataset` into this SQL database.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import _benchmark_dset_readtime  # NOQA
>>> import kwcoco
>>> from kwcoco.coco_sql_dataset import *
>>> dset2 = dset = kwcoco.CocoDataset.demo()
>>> dset1 = self = CocoSqlDatabase('sqlite:///:memory:')
>>> self.connect()
>>> self.populate_from(dset)
>>> assert_dsets_allclose(dset1, dset2, tag1='sql', tag2='dct')
>>> ti_sql = _benchmark_dset_readtime(dset1, 'sql')
>>> ti_dct = _benchmark_dset_readtime(dset2, 'dct')
>>> print('ti_sql.rankings = {}'.format(ub.repr2(ti_sql.rankings, nl=2,␣
↪precision=6, align=':')))
>>> print('ti_dct.rankings = {}'.format(ub.repr2(ti_dct.rankings, nl=2,␣
↪precision=6, align=':')))
```

property dataset(*self*)

property anns(*self*)

property cats(*self*)

property imgs(*self*)

property name_to_cat(*self*)

raw_table(*self*, *table_name*)
Loads an entire SQL table as a pandas DataFrame

Parameters **table_name** (*str*) – name of the table

Returns DataFrame

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo()
>>> table_df = self.raw_table('annotations')
>>> print(table_df)
```

_column_lookup(*self*, *tablename*, *key*, *rowids*, *default=ub.NoParam*, *keepid=False*)

Convinience method to lookup only a single column of information

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo(10)
>>> tablename = 'annotations'
>>> key = 'category_id'
>>> rowids = list(self.anns.keys())[::3]
>>> cids1 = self._column_lookup(tablename, key, rowids)
>>> cids2 = self.annots(rowids).get(key)
>>> cids3 = dset.annots(rowids).get(key)
>>> assert cids3 == cids2 == cids1
```

Ignore: import timerit ti = timerit.Timerit(10, bestof=3, verbose=2)

for timer in ti.reset('time'):

with timer: self._column_lookup(tablename, key, rowids)

for timer in ti.reset('time'): self.anns._cache.clear() with timer:

annots = self.annots(rowids) annots.get(key)

for timer in ti.reset('time'): self.anns._cache.clear() with timer:

anns = [self.anns[aid] for aid in rowids] cids = [ann[key] for ann in anns]

_all_rows_column_lookup(*self*, *tablename*, *keys*)

Convinience method to look up all rows from a table and only a few columns.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo(10)
>>> tablename = 'annotations'
>>> keys = ['id', 'category_id']
>>> rows = self._all_rows_column_lookup(tablename, keys)
```

tabular_targets(*self*)

Convinience method to create an in-memory summary of basic annotation properties with minimal SQL overhead.

Example

```
>>> # xdoctest: +REQUIRES(module:sqlalchemy)
>>> from kwcoco.coco_sql_dataset import *  # NOQA
>>> self, dset = demo()
>>> targets = self.tabular_targets()
>>> print(targets.pandas())
```

property bundle_dpath(*self*)

property data_fpath(*self*)
data_fpath is an alias of fpath

classmethod coerce(*self*, *data*)
Create an SQL CocoDataset from the input pointer.

Example

import kwcoco dset = kwcoco.CocoDataset.demo('shapes8') data = dset.fpath self = CocoSqlDatabase.coerce(data)

from kwcoco.coco_sql_dataset import CocoSqlDatabase import kwcoco dset = kwcoco.CocoDataset.coerce('spacenet7.kwcoco.json')

self = CocoSqlDatabase.coerce(dset)

from kwcoco.coco_sql_dataset import CocoSqlDatabase sql_dset = CocoSqlDatabase.coerce('spacenet7.kwcoco.json')

from kwcoco.coco_sql_dataset import CocoSqlDatabase import kwcoco sql_dset = kwcoco.CocoDataset.coerce('_spacenet7.kwcoco.view.v006.sqlite')

kwcoco.coco_sql_dataset.**cached_sql_coco_view**(*dct_db_fpath=None*, *sql_db_fpath=None*, *dset=None*, *force_rewrite=False*)
Attempts to load a cached SQL-View dataset, only loading and converting the json dataset if necessary.

kwcoco.coco_sql_dataset.**ensure_sql_coco_view**(*dset*, *db_fpath=None*, *force_rewrite=False*)
Create a cached on-disk SQL view of an on-disk COCO dataset.

Note: This function is fragile. It depends on looking at file modified timestamps to determine if it needs to write the dataset.

---

kwcoco.coco_sql_dataset.**demo**(*num=10*)

kwcoco.coco_sql_dataset.**assert_dsets_allclose**(*dset1*, *dset2*, *tag1='dset1'*, *tag2='dset2'*)

kwcoco.coco_sql_dataset.**_benchmark_dset_readtime**(*dset*, *tag='?'*)
Helper for understanding the time differences between backends

kwcoco.coco_sql_dataset.**_benchmark_dict_proxy_ops**(*proxy*)
Get insight on the efficiency of operations

kwcoco.coco_sql_dataset.**devcheck**()
Scratch work for things that should eventually become unit or doc tests

from kwcoco.coco_sql_dataset import * # NOQA self, dset = demo()

2.3.11 kwcoco.compat_dataset

A wrapper around the basic kwcoco dataset with a pycocotools API.

We do not recommend using this API because it has some idiosyncrasies, where names can be missleading and APIs are not always clear / efficient: e.g.

(1) catToImgs returns integer image ids but imgToAnns returns annotation dictionaries.

(2) showAnns takes a dictionary list as an argument instead of an integer list

The cool thing is that this extends the kwcoco API so you can drop this for compatibility with the old API, but you still get access to all of the kwcoco API including dynamic addition / removal of categories / annotations / images.

Module Contents

Classes

COCO	A wrapper around the basic kwcoco dataset with a pycocotools API.

class kwcoco.compat_dataset.**COCO**(*annotation_file=None*, ***kw*)

Bases: *kwcoco.coco_dataset.CocoDataset*

A wrapper around the basic kwcoco dataset with a pycocotools API.

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> basic = kwcoco.CocoDataset.demo('shapes8')
>>> self = COCO(basic.dataset)
>>> self.info()
>>> print('self.imgToAnns = {!r}'.format(self.imgToAnns[1]))
>>> print('self.catToImgs = {!r}'.format(self.catToImgs))
```

createIndex(*self*)

info(*self*)

Print information about the annotation file. :return:

property imgToAnns(*self*)

property catToImgs(*self*)

unlike the name implies, this actually goes from category to image ids Name retained for backward compatibility

getAnnIds(*self*, *imgIds=[]*, *catIds=[]*, *areaRng=[]*, *iscrowd=None*)

Get ann ids that satisfy given filter conditions. default skips that filter :param imgIds (int array) : get anns for given imgs

catIds (int array) : get anns for given cats areaRng (float array) : get anns for given area range (e.g. [0 inf]) iscrowd (boolean) : get anns for given crowd label (False or True)

Returns ids (int array) : integer array of ann ids

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> self.getAnnIds()
>>> self.getAnnIds(imgIds=1)
>>> self.getAnnIds(imgIds=[1])
>>> self.getAnnIds(catIds=[3])
```

getCatIds(*self*, *catNms=[]*, *supNms=[]*, *catIds=[]*)

filtering parameters. default skips that filter. :param catNms (str array) : get cats for given cat names :param supNms (str array) : get cats for given supercategory names :param catIds (int array) : get cats for given cat ids :return: ids (int array) : integer array of cat ids

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> self.getCatIds()
>>> self.getCatIds(catNms=['superstar'])
>>> self.getCatIds(supNms=['raster'])
>>> self.getCatIds(catIds=[3])
```

getImgIds(*self*, *imgIds=[]*, *catIds=[]*)

Get img ids that satisfy given filter conditions. :param imgIds (int array) : get imgs for given ids :param catIds (int array) : get imgs with all given cats :return: ids (int array) : integer array of img ids

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> self.getImgIds(imgIds=[1, 2])
>>> self.getImgIds(catIds=[3, 6, 7])
>>> self.getImgIds(catIds=[3, 6, 7], imgIds=[1, 2])
```

loadAnns(*self*, *ids=[]*)

Load anns with the specified ids. :param ids (int array) : integer ids specifying anns :return: anns (object array) : loaded ann objects

loadCats(*self*, *ids=[]*)

Load cats with the specified ids. :param ids (int array) : integer ids specifying cats :return: cats (object array) : loaded cat objects

loadImgs(*self*, *ids=[]*)

Load anns with the specified ids. :param ids (int array) : integer ids specifying img :return: imgs (object array) : loaded img objects

showAnns(*self*, *anns*, *draw_bbox=False*)

Display the specified annotations. :param anns (array of object): annotations to display :return: None

loadRes(*self*, *resFile*)
Load result file and return a result api object. :param resFile (str) : file name of result file :return: res (obj) : result api object

download(*self*, *tarDir=None*, *imgIds=[]*)
Download COCO images from mscoco.org server. :param tarDir (str): COCO results directory name

imgIds (list): images to be downloaded

Returns

loadNumpyAnnotations(*self*, *data*)
Convert result data from a numpy array [Nx7] where each row contains {imageID,x1,y1,w,h,score,class} :param data (numpy.ndarray) :return: annotations (python nested list)

annToRLE(*self*, *ann*)
Convert annotation which can be polygons, uncompressed RLE to RLE. :return: binary mask (numpy 2D array)

Notes

- This requires the C-extensions for kwimage to be installed due to the need to interface with the bytes RLE format.

Example

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> try:
>>>     rle = self.annToRLE(self.anns[1])
>>> except NotImplementedError:
>>>     import pytest
>>>     pytest.skip('missing kwimage c-extensions')
>>> else:
>>>     assert len(rle['counts']) > 2
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> self.conform(legacy=True)
>>> orig = self._aspycoco().annToRLE(self.anns[1])
```

annToMask(*self*, *ann*)
Convert annotation which can be polygons, uncompressed RLE, or RLE to binary mask.

Returns binary mask (numpy 2D array)

Notes

The mask is returned as a fortran (F-style) array with the same dimensions as the parent image.

Ignore:

```
>>> from kwcoco.compat_dataset import *  # NOQA
>>> import kwcoco
>>> self = COCO(kwcoco.CocoDataset.demo('shapes8').dataset)
>>> mask = self.annToMask(self.anns[1])
>>> # xdoctest: +REQUIRES(module:pycocotools)
>>> self.conform(legacy=True)
>>> orig = self._aspycoco().annToMask(self.anns[1])
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> diff = kwimage.normalize((compat_mask - orig_mask).astype(np.float32))
>>> kwplot.imshow(diff)
>>> kwplot.show_if_requested()
```

2.3.12 kwcoco.kpf

WIP:

Conversions to and from KPF format.

Module Contents

Functions

coco_to_kpf(coco_dset)	import kwcoco
demo()	

kwcoco.kpf.**coco_to_kpf**(*coco_dset*)
import kwcoco coco_dset = kwcoco.CocoDataset.demo('shapes8')

kwcoco.kpf.**demo**()

2.3.13 kwcoco.kw18

A helper for converting COCO to / from KW18 format.

KW18 File Format https://docs.google.com/spreadsheets/d/1DFCwoTKnDv8qfy3raM7QXtir2Fjfj9j8-z8px5Bu0q8/edit#gid=10

The kw18.trk files are text files, space delimited; each row is one frame of one track and all rows have the same number of columns. The fields are:

01) track_ID : identifies the track 02) num_frames: number of frames in the track 03) frame_id : frame number for this track sample 04) loc_x : X-coordinate of the track (image/ground coords) 05) loc_y : Y-coordinate of the track (image/ground coords) 06) vel_x : X-velocity of the object (image/ground coords) 07) vel_y : Y-velocity of the object (image/ground coords) 08) obj_loc_x : X-coordinate of the object (image coords) 09) obj_loc_y : Y-coordinate of the

object (image coords) 10) bbox_min_x : minimum X-coordinate of bounding box (image coords) 11) bbox_min_y : minimum Y-coordinate of bounding box (image coords) 12) bbox_max_x : maximum X-coordinate of bounding box (image coords) 13) bbox_max_y : maximum Y-coordinate of bounding box (image coords) 14) area : area of object (pixels) 15) world_loc_x : X-coordinate of object in world 16) world_loc_y : Y-coordinate of object in world 17) world_loc_z : Z-coordiante of object in world 18) timestamp : timestamp of frame (frames) For the location and velocity of object centroids, use fields 4-7. Bounding box is specified using coordinates of the top-left and bottom right corners. Fields 15-17 may be ignored.

The kw19.trk and kw20.trk files, when present, add the following field(s): 19) object class: estimated class of the object, either 1 (person), 2 (vehicle), or 3 (other). 20) Activity ID – refer to activities.txt for index and list of activities.

Module Contents

Classes

KW18	A DataFrame like object that stores KW18 column data

Functions

_ensure_kw18_column_order(df)	Ensure expected kw18 columns exist and are in the correct order.

class kwcoco.kw18.**KW18**(*data*)

Bases: kwarray.DataFrameArray

A DataFrame like object that stores KW18 column data

Example

```
>>> import kwcoco
>>> from kwcoco.kw18 import KW18
>>> coco_dset = kwcoco.CocoDataset.demo('shapes')
>>> kw18_dset = KW18.from_coco(coco_dset)
>>> print(kw18_dset.pandas())
```

DEFAULT_COLUMNS = ['track_id', 'track_length', 'frame_number', 'tracking_plane_loc_x', 'tracking_plane_loc_y',...

classmethod demo(*KW18*)

classmethod from_coco(*KW18*, *coco_dset*)

to_coco(*self*, *image_paths=None*, *video_name=None*)

Translates a kw18 files to a CocoDataset.

Note: kw18 does not contain complete information, and as such the returned coco dataset may need to be augmented.

Parameters

- **image_paths** (*Dict[int, str], default=None*) – if specified, maps frame numbers to image file paths.
- **video_name** (*str, default=None*) – if specified records the name of the video this kw18 belongs to

Todo:

- [X] allow kwargs to specify path to frames / videos

Example

```
>>> from kwcoco.kw18 import KW18
>>> from os.path import join
>>> import ubelt as ub
>>> import kwimage
>>> # Prep test data - autogen a demo kw18 and write it to disk
>>> dpath = ub.ensure_app_cache_dir('kwcoco/kw18')
>>> kw18_fpath = join(dpath, 'test.kw18')
>>> KW18.demo().dump(kw18_fpath)
>>> #
>>> # Load the kw18 file
>>> self = KW18.load(kw18_fpath)
>>> # Pretend that these image correspond to kw18 frame numbers
>>> frame_names = [kwimage.grab_test_image_fpath(k) for k in kwimage.grab_test_
↪image.keys()]
>>> frame_ids = sorted(set(self['frame_number']))
>>> image_paths = dict(zip(frame_ids, frame_names))
>>> #
>>> # Convert the kw18 to kwcoco and specify paths to images
>>> coco_dset = self.to_coco(image_paths=image_paths, video_name='dummy.mp4')
>>> #
>>> # Now we can draw images
>>> canvas = coco_dset.draw_image(1)
>>> # xdoctest: +REQUIRES(--draw)
>>> kwimage.imwrite('foo.jpg', canvas)
>>> # Draw all iamges
>>> for gid in coco_dset.imgs.keys():
>>>     canvas = coco_dset.draw_image(gid)
>>>     fpath = join(dpath, 'gid_{}.jpg'.format(gid))
>>>     print('write fpath = {!r}'.format(fpath))
>>>     kwimage.imwrite(fpath, canvas)
```

classmethod load(*KW18*, *file*)

Example

```
>>> import kwcoco
>>> from kwcoco.kw18 import KW18
>>> coco_dset = kwcoco.CocoDataset.demo('shapes')
>>> kw18_dset = KW18.from_coco(coco_dset)
>>> print(kw18_dset.pandas())
```

classmethod loads(*KW18*, *text*)

Example

```
>>> self = KW18.demo()
>>> text = self.dumps()
>>> self2 = KW18.loads(text)
>>> empty = KW18.loads('')
```

dump(*self*, *file*)

dumps(*self*)

Example

```
>>> self = KW18.demo()
>>> text = self.dumps()
>>> print(text)
```

kwcoco.kw18.**_ensure_kw18_column_order**(*df*)

Ensure expected kw18 columns exist and are in the correct order.

Example

```
>>> import pandas as pd
>>> df = pd.DataFrame(columns=KW18.DEFAULT_COLUMNS[0:18])
>>> _ensure_kw18_column_order(df)
>>> df = pd.DataFrame(columns=KW18.DEFAULT_COLUMNS[0:19])
>>> _ensure_kw18_column_order(df)
>>> df = pd.DataFrame(columns=KW18.DEFAULT_COLUMNS[0:18] + KW18.DEFAULT_
↪COLUMNS[20:21])
>>> assert np.all(_ensure_kw18_column_order(df).columns == df.columns)
```

2.4 Package Contents

2.4.1 Classes

AbstractCocoDataset	This is a common base for all variants of the Coco Dataset
CategoryTree	Wrapper that maintains flat or hierarchical category information.
CocoDataset	

Notes

class kwcoco.**AbstractCocoDataset**

Bases: abc.ABC

This is a common base for all variants of the Coco Dataset

At the time of writing there is kwcoco.CocoDataset (which is the dictionary-based backend), and the kwcoco.coco_sql_dataset.CocoSqlDataset, which is experimental.

class kwcoco.**CategoryTree**(*graph=None*)

Bases: ubelt.NiceRepr

Wrapper that maintains flat or hierarchical category information.

Helps compute softmaxes and probabilities for tree-based categories where a directed edge (A, B) represents that A is a superclass of B.

Notes

There are three basic properties that this object maintains:

node: Alphanumeric string names that should be generally descriptive. Using spaces and special characters in these names is discouraged, but can be done. This is the COCO category "name" attribute. For categories this may be denoted as (name, node, cname, catname).

id: The integer id of a category should ideally remain consistent. These are often given by a dataset (e.g. a COCO dataset). This is the COCO category "id" attribute. For categories this is often denoted as (id, cid).

index: Contigous zero-based indices that indexes the list of categories. These should be used for the fastest access in backend computation tasks. Typically corresponds to the ordering of the channels in the final linear layer in an associated model. For categories this is often denoted as (index, cidx, idx, or cx).

Variables

- **idx_to_node** (*List[str]*) – a list of class names. Implicitly maps from index to category name.
- **id_to_node** (*Dict[int, str]*) – maps integer ids to category names
- **node_to_id** (*Dict[str, int]*) – maps category names to ids
- **node_to_idx** (*Dict[str, int]*) – maps category names to indexes

- **graph** (*nx.Graph*) – a Graph that stores any hierarchy information. For standard mutually exclusive classes, this graph is edgeless. Nodes in this graph can maintain category attributes / properties.
- **idx_groups** (*List[List[int]]*) – groups of category indices that share the same parent category.

Example

```
>>> from kwcoco.category_tree import *
>>> graph = nx.from_dict_of_lists({
>>>     'background': [],
>>>     'foreground': ['animal'],
>>>     'animal': ['mammal', 'fish', 'insect', 'reptile'],
>>>     'mammal': ['dog', 'cat', 'human', 'zebra'],
>>>     'zebra': ['grevys', 'plains'],
>>>     'grevys': ['fred'],
>>>     'dog': ['boxer', 'beagle', 'golden'],
>>>     'cat': ['maine coon', 'persian', 'sphynx'],
>>>     'reptile': ['bearded dragon', 't-rex'],
>>> }, nx.DiGraph)
>>> self = CategoryTree(graph)
>>> print(self)
<CategoryTree(nNodes=22, maxDepth=6, maxBreadth=4...)>
```

Example

```
>>> # The coerce classmethod is the easiest way to create an instance
>>> import kwcoco
>>> kwcoco.CategoryTree.coerce(['a', 'b', 'c'])
<CategoryTree(nNodes=3, nodes=['a', 'b', 'c']) ...
>>> kwcoco.CategoryTree.coerce(4)
<CategoryTree(nNodes=4, nodes=['class_1', 'class_2', 'class_3', ...
>>> kwcoco.CategoryTree.coerce(4)
```

copy(*self*)

classmethod from_mutex(*cls*, *nodes*, *bg_hack=True*)

Parameters **nodes** (*List[str]*) – or a list of class names (in which case they will all be assumed to be mutually exclusive)

Example

```
>>> print(CategoryTree.from_mutex(['a', 'b', 'c']))
<CategoryTree(nNodes=3, ...)>
```

classmethod from_json(*cls*, *state*)

Parameters **state** (*Dict*) – see __getstate__ / __json__ for details

classmethod from_coco(*cls*, *categories*)

Create a CategoryTree object from coco categories

Parameters **List[Dict]** – list of coco-style categories

classmethod coerce(*cls*, *data*, ***kw*)

Attempt to coerce data as a CategoryTree object.

This is primarily useful for when the software stack depends on categories being represent

This will work if the input data is a specially formatted json dict, a list of mutually exclusive classes, or if it is already a CategoryTree. Otherwise an error will be thrown.

Parameters

- **data** (*object*) – a known representation of a category tree.
- ****kwargs** – input type specific arguments

Returns self

Return type *CategoryTree*

Raises

- **TypeError - if the input format is unknown** –
- **ValueError - if kwargs are not compatible with the input format** –

Example

```
>>> import kwcoco
>>> classes1 = kwcoco.CategoryTree.coerce(3)  # integer
>>> classes2 = kwcoco.CategoryTree.coerce(classes1.__json__())  # graph dict
>>> classes3 = kwcoco.CategoryTree.coerce(['class_1', 'class_2', 'class_3'])  #␣
↪mutex list
>>> classes4 = kwcoco.CategoryTree.coerce(classes1.graph)  # nx Graph
>>> classes5 = kwcoco.CategoryTree.coerce(classes1)  # cls
>>> # xdoctest: +REQUIRES(module:ndsampler)
>>> import ndsampler
>>> classes6 = ndsampler.CategoryTree.coerce(3)
>>> classes7 = ndsampler.CategoryTree.coerce(classes1)
>>> classes8 = kwcoco.CategoryTree.coerce(classes6)
```

classmethod demo(*cls*, *key='coco'*, ***kwargs*)

Parameters **key** (*str*) – specify which demo dataset to use. Can be ‘coco’ (which uses the default coco demo data). Can be ‘btree’ which creates a binary tree and accepts kwargs

‘r’ and ‘h’ for branching-factor and height.

Can be 'btree2', which is the same as btree but returns strings

CommandLine: xdoctest -m ~/code/kwcoco/kwcoco/category_tree.py CategoryTree.demo

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self = {}'.format(self))
self = <CategoryTree(nNodes=10, maxDepth=2, maxBreadth=4...)>
```

to_coco(*self*)

Converts to a coco-style data structure

Yields *Dict* – coco category dictionaries

id_to_idx(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.id_to_idx[1]
```

idx_to_id(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CategoryTree.demo()
>>> self.idx_to_id[0]
```

idx_to_ancestor_idxs(*self*, *include_self=True*)

Mapping from a class index to its ancestors

Parameters **include_self** (*bool, default=True*) – if True includes each node as its own ancestor.

idx_to_descendants_idxs(*self*, *include_self=False*)

Mapping from a class index to its descendants (including itself)

Parameters **include_self** (*bool, default=False*) – if True includes each node as its own descendant.

idx_pairwise_distance(*self*)

Get a matrix encoding the distance from one class to another.

Distances

- from parents to children are positive (descendants),
- from children to parents are negative (ancestors),
- between unreachable nodes (wrt to forward and reverse graph) are

nan.

__len__(*self*)

__iter__(*self*)

__getitem__(*self*, *index*)

__contains__(*self*, *node*)

__json__(*self*)

Example

```
>>> import pickle
>>> self = CategoryTree.demo()
>>> print('self = {!r}'.format(self.__json__()))
```

__getstate__(*self*)

Serializes information in this class

Example

```
>>> from kwcoco.category_tree import *
>>> import pickle
>>> self = CategoryTree.demo()
>>> state = self.__getstate__()
>>> serialization = pickle.dumps(self)
>>> recon = pickle.loads(serialization)
>>> assert recon.__json__() == self.__json__()
```

__setstate__(*self*, *state*)

__nice__(*self*)

is_mutex(*self*)

Returns True if all categories are mutually exclusive (i.e. flat)

If true, then the classes may be represented as a simple list of class names without any loss of information, otherwise the underlying category graph is necessary to preserve all knowledge.

Todo:

- [ ] what happens when we have a dummy root?

---

property num_classes(*self*)

property class_names(*self*)

property category_names(*self*)

property cats(*self*)

Returns a mapping from category names to category attributes.

If this category tree was constructed from a coco-dataset, then this will contain the coco category attributes.

Returns Dict[str, Dict[str, object]]

Example

```
>>> from kwcoco.category_tree import *
>>> self = CategoryTree.demo()
>>> print('self.cats = {!r}'.format(self.cats))
```

index(*self*, *node*)
Return the index that corresponds to the category name

_build_index(*self*)
construct lookup tables

show(*self*)

Ignore:

```
>>> import kwplot
>>> kwplot.autompl()
>>> from kwcoco import category_tree
>>> self = category_tree.CategoryTree.demo()
>>> self.show()
```

python -c "import kwplot, kwcoco, graphid; kwplot.autompl(); graphid.util.show_nx(kwcoco.category_tree.CategoryTree.demo().graph); kwplot.show_if_requested()" --show

class kwcoco.**CocoDataset**(*data=None*, *tag=None*, *bundle_dpath=None*, *img_root=None*, *fname=None*, *autobuild=True*)
Bases: *kwcoco.abstract_coco_dataset.AbstractCocoDataset*, MixinCocoAddRemove, MixinCocoStats, MixinCocoObjects, MixinCocoDraw, MixinCocoAccessors, MixinCocoExtras, MixinCocoIndex, MixinCocoDepricate, ubelt.NiceRepr

Notes

A keypoint annotation

{ "image_id" : int, "category_id" : int, "keypoints" : [x1,y1,v1,...,xk,yk,vk], "score" : float,

} Note that `v[i]` is a visibility flag, where v=0: not labeled,

v=1: labeled but not visible, and v=2: labeled and visible.

A bounding box annotation

{ "image_id" : int, "category_id" : int, "bbox" : [x,y,width,height], "score" : float,

}

We also define a non-standard "line" annotation (which our fixup scripts will interpret as the diameter of a circle to convert into a bounding box)

A line* annotation (note this is a non-standard field)

{ "image_id" : int, "category_id" : int, "line" : [x1,y1,x2,y2], "score" : float,

}

Lastly, note that our datasets will sometimes specify multiple bbox, line, and/or, keypoints fields. In this case we may also specify a field roi_shape, which denotes which field is the "main" annotation type.

Variables

- **dataset** (*Dict*) – raw json data structure. This is the base dictionary that contains {'annotations': List, 'images': List, 'categories': List}
- **index** (CocoIndex) – an efficient lookup index into the coco data structure. The index defines its own attributes like `anns`, `cats`, `imgs`, etc. See `CocoIndex` for more details on which attributes are available.
- **fpath** (*PathLike | None*) – if known, this stores the filepath the dataset was loaded from
- **tag** (*str*) – A tag indicating the name of the dataset.
- **bundle_dpath** (*PathLike | None*) – If known, this is the root path that all image file names are relative to. This can also be manually overwritten by the user.
- **hashid** (*str | None*) – If computed, this will be a hash uniquely identifing the dataset. To ensure this is computed see `_build_hashid()`.

References

http://cocodataset.org/#format http://cocodataset.org/#download

CommandLine: python -m kwcoco.coco_dataset CocoDataset –show

Example

```
>>> dataset = demo_coco_data()
>>> self = CocoDataset(dataset, tag='demo')
>>> # xdoctest: +REQUIRES(--show)
>>> self.show_image(gid=2)
>>> from matplotlib import pyplot as plt
>>> plt.show()
```

property fpath(*self*)
In the future we will deprecate img_root for bundle_dpath

_infer_dirs(*self*)

Example

self = dset

classmethod from_data(*CocoDataset*, *data*, *bundle_dpath=None*, *img_root=None*)
Constructor from a json dictionary

classmethod from_image_paths(*CocoDataset*, *gpaths*, *bundle_dpath=None*, *img_root=None*)
Constructor from a list of images paths.

This is a convinience method.

Parameters **gpaths** (*List[str]*) – list of image paths

Example

```
>>> coco_dset = CocoDataset.from_image_paths(['a.png', 'b.png'])
>>> assert coco_dset.n_images == 2
```

classmethod from_coco_paths(*CocoDataset*, *fpaths*, *max_workers=0*, *verbose=1*, *mode='thread'*, *union='try'*)

Constructor from multiple coco file paths.

Loads multiple coco datasets and unions the result

Notes

if the union operation fails, the list of individually loaded files is returned instead.

Parameters

- **fpaths** (*List[str]*) – list of paths to multiple coco files to be loaded and unioned.
- **max_workers** (*int, default=0*) – number of worker threads / processes
- **verbose** (*int*) – verbosity level
- **mode** (*str*) – thread, process, or serial
- **union** (*str | bool, default='try'*) – If True, unions the result datasets after loading. If False, just returns the result list. If 'try', then try to preform the union, but return the result list if it fails.

copy(*self*)

Deep copies this object

Example

```
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> new = self.copy()
>>> assert new.imgs[1] is new.dataset['images'][0]
>>> assert new.imgs[1] == self.dataset['images'][0]
>>> assert new.imgs[1] is not self.dataset['images'][0]
```

__nice__(*self*)

dumps(*self*, *indent=None*, *newlines=False*)

Writes the dataset out to the json format

Parameters **newlines** (*bool*) – if True, each annotation, image, category gets its own line

Notes

Using newlines=True is similar to: print(ub.repr2(dset.dataset, nl=2, trailsep=False)) However, the above may not output valid json if it contains ndarrays.

Example

```
>>> from kwcoco.coco_dataset import *
>>> import json
>>> self = CocoDataset.demo()
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> text = self.dumps(newlines=True)
>>> print(text)
>>> self2 = CocoDataset(json.loads(text), tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

dump(*self*, *file*, *indent=None*, *newlines=False*)
Writes the dataset out to the json format

Parameters

- **file** (*PathLike | FileLike*) – Where to write the data. Can either be a path to a file or an open file pointer / stream.
- **newlines** (*bool*) – if True, each annotation, image, category gets its own line.

Example

```
>>> import tempfile
>>> from kwcoco.coco_dataset import *
>>> self = CocoDataset.demo()
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = CocoDataset(dataset, tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

```
>>> file = tempfile.NamedTemporaryFile('w')
>>> self.dump(file, newlines=True)
>>> file.seek(0)
>>> text = open(file.name, 'r').read()
```

(continues on next page)

(continued from previous page)

```
>>> print(text)
>>> file.seek(0)
>>> dataset = json.load(open(file.name, 'r'))
>>> self2 = CocoDataset(dataset, tag='demo2')
>>> assert self2.dataset == self.dataset
>>> assert self2.dataset is not self.dataset
```

_check_json_serializable(*self*, *verbose=1*)
Debug which part of a coco dataset might not be json serializable

_check_integrity(*self*)
perform all checks

_check_index(*self*)

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo()
>>> self._check_index()
>>> # Force a failure
>>> self.index.anns.pop(1)
>>> self.index.anns.pop(2)
>>> import pytest
>>> with pytest.raises(AssertionError):
>>>     self._check_index()
```

_check_pointers(*self*, *verbose=1*)
Check that all category and image ids referenced by annotations exist

_build_index(*self*)

union(**others*, *disjoint_tracks=True*, ***kwargs*)
Merges multiple *CocoDataset* items into one. Names and associations are retained, but ids may be different.

Parameters

- ***others** – a series of CocoDatasets that we will merge. Note, if called as an instance method, the "self" instance will be the first item in the "others" list. But if called like a classmethod, "others" will be empty by default.
- **disjoint_tracks** (*bool, default=True*) – if True, we will assume track-ids are disjoint and if two datasets share the same track-id, we will disambiguate them. Otherwise they will be copied over as-is.
- ****kwargs** – constructor options for the new merged CocoDataset

Returns a new merged coco dataset

Return type *CocoDataset*

CommandLine: xdoctest -m kwcoco.coco_dataset CocoDataset.union

Example

```
>>> # Test union works with different keypoint categories
>>> dset1 = CocoDataset.demo('shapes1')
>>> dset2 = CocoDataset.demo('shapes2')
>>> dset1.remove_keypoint_categories(['bot_tip', 'mid_tip', 'right_eye'])
>>> dset2.remove_keypoint_categories(['top_tip', 'left_eye'])
>>> dset_12a = CocoDataset.union(dset1, dset2)
>>> dset_12b = dset1.union(dset2)
>>> dset_21 = dset2.union(dset1)
>>> def add_hist(h1, h2):
>>>     return {k: h1.get(k, 0) + h2.get(k, 0) for k in set(h1) | set(h2)}
>>> kpfreq1 = dset1.keypoint_annotation_frequency()
>>> kpfreq2 = dset2.keypoint_annotation_frequency()
>>> kpfreq_want = add_hist(kpfreq1, kpfreq2)
>>> kpfreq_got1 = dset_12a.keypoint_annotation_frequency()
>>> kpfreq_got2 = dset_12b.keypoint_annotation_frequency()
>>> assert kpfreq_want == kpfreq_got1
>>> assert kpfreq_want == kpfreq_got2
```

```
>>> # Test disjoint gid datasets
>>> import kwcoco
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> for new_gid, img in enumerate(dset1.dataset['images'], start=10):
>>>     for aid in dset1.gid_to_aids[img['id']]:
>>>         dset1.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset1._build_index()
>>> # ------
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> for new_gid, img in enumerate(dset2.dataset['images'], start=100):
>>>     for aid in dset2.gid_to_aids[img['id']]:
>>>         dset2.anns[aid]['image_id'] = new_gid
>>>     img['id'] = new_gid
>>> dset1.index.clear()
>>> dset2._build_index()
>>> others = [dset1, dset2]
>>> merged = kwcoco.CocoDataset.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([10, 11, 12, 100, 101]) == set(merged.imgs)
```

```
>>> # Test data is not preserved
>>> dset2 = kwcoco.CocoDataset.demo('shapes2')
>>> dset1 = kwcoco.CocoDataset.demo('shapes3')
>>> others = (dset1, dset2)
>>> cls = self = kwcoco.CocoDataset
>>> merged = cls.union(*others)
>>> print('merged = {!r}'.format(merged))
>>> print('merged.imgs = {}'.format(ub.repr2(merged.imgs, nl=1)))
>>> assert set(merged.imgs) & set([1, 2, 3, 4, 5]) == set(merged.imgs)
```

```
>>> # Test track-ids are mapped correctly
>>> dset1 = kwcoco.CocoDataset.demo('vidshapes1')
>>> dset2 = kwcoco.CocoDataset.demo('vidshapes2')
>>> dset3 = kwcoco.CocoDataset.demo('vidshapes3')
>>> others = (dset1, dset2, dset3)
>>> for dset in others:
>>>     [a.pop('segmentation', None) for a in dset.index.anns.values()]
>>>     [a.pop('keypoints', None) for a in dset.index.anns.values()]
>>> cls = self = kwcoco.CocoDataset
>>> merged = cls.union(*others, disjoint_tracks=1)
>>> print('dset1.anns = {}'.format(ub.repr2(dset1.anns, nl=1)))
>>> print('dset2.anns = {}'.format(ub.repr2(dset2.anns, nl=1)))
>>> print('dset3.anns = {}'.format(ub.repr2(dset3.anns, nl=1)))
>>> print('merged.anns = {}'.format(ub.repr2(merged.anns, nl=1)))
```

Example

```
>>> import kwcoco
>>> # Test empty union
>>> empty_union = kwcoco.CocoDataset.union()
>>> assert len(empty_union.index.imgs) == 0
```

Todo:

- [ ] are supercategories broken?
- [ ] reuse image ids where possible
- [ ] reuse annotation / category ids where possible
- [X] handle case where no inputs are given
- [x] disambiguate track-ids
- [x] disambiguate video-ids

---

subset(*self*, *gids*, *copy=False*, *autobuild=True*)

Return a subset of the larger coco dataset by specifying which images to port. All annotations in those images will be taken.

Parameters

- **gids** (*List[int]*) – image-ids to copy into a new dataset
- **copy** (*bool, default=False*) – if True, makes a deep copy of all nested attributes, otherwise makes a shallow copy.
- **autobuild** (*bool, default=True*) – if True will automatically build the fast lookup index.

Example

```
>>> self = CocoDataset.demo()
>>> gids = [1, 3]
>>> sub_dset = self.subset(gids)
>>> assert len(self.index.gid_to_aids) == 3
>>> assert len(sub_dset.gid_to_aids) == 2
```

Example

```
>>> import kwcoco
>>> self = kwcoco.CocoDataset.demo('vidshapes2')
>>> gids = [1, 2]
>>> sub_dset = self.subset(gids, copy=True)
>>> assert len(sub_dset.index.videos) == 1
>>> assert len(self.index.videos) == 2
```

Example

```
>>> self = CocoDataset.demo()
>>> sub1 = self.subset([1])
>>> sub2 = self.subset([2])
>>> sub3 = self.subset([3])
>>> others = [sub1, sub2, sub3]
>>> rejoined = CocoDataset.union(*others)
>>> assert len(sub1.anns) == 9
>>> assert len(sub2.anns) == 2
>>> assert len(sub3.anns) == 0
>>> assert rejoined.basic_stats() == self.basic_stats()
```

view_sql(*self*, *force_rewrite=False*, *memory=False*)

Create a cached SQL interface to this dataset suitable for large scale multiprocessing use cases.

Parameters

- **force_rewrite** (*bool, default=False*) – if True, forces an update to any existing cache file on disk
- **memory** (*bool, default=False*) – if True, the database is constructed in memory.

Note: This view cache is experimental and currently depends on the timestamp of the file pointed to by `self.fpath`. In other words dont use this on in-memory datasets.

PYTHON MODULE INDEX

k

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

W

X