# Labeltool Documentation

***Release 0.1***

**cv:hci lab, Institute for Anthropomatics, Karlsruhe Institute of Tec**

January 07, 2017

Contents

This is the documentation of Sloth. Sloth's purpose is to provide a versatile tool for various labeling tasks in the context of computer vision research. Since there are so many different label formats and requirements out there, we concluded that is virtually impossible to build *the one* label tool sufficient to handle all labeling tasks. Therefore, this project can be seen rather as a framework and set of standard components to quickly configure a label tool specificly tailored to ones needs.

In this documentation we will go over some of the key concepts of Sloth, how to configure Sloth using the standard components provided in the package, and finally how to go further and write custom visualization items and label format containers to deal with specific labeling needs.

CHAPTER 1

# Feedback

*Please* provide feedback to us on this document and Sloth in general! We won't be able to incorporate your required features if you do not talk to us. Also, use the bug tracker at https://github.com/cvhciKIT/sloth/issues. Of course, similarly welcome are patches!

CHAPTER 2

# Contents

## 2.1 Installation Guide

Before you can install Sloth, make sure that you have all the prerequisites installed.

### 2.1.1 Prerequisites

Sloth is implemented in Python and PyQt4, so it needs both. It further depends on either PIL or okapy for image loading.

To use okapy, make sure to make it's modules known to python, e.g. add <okapibuild>/python/ to the PYTHONPATH environment variable:

```
export PYTHONPATH=<okapibuild>/python/:$PYTHONPATH
```

For compiling the docs, Python Sphinx is needed.

### 2.1.2 Installing Sloth

Run with administrator priviledges:

```
python setup.py install
```

## 2.2 Concepts

In this section, we will introduce some high-level concepts of Sloth.

### 2.2.1 Labels

Sloth is designed for labeling a set of images or videos. Each image, or video frame, can contain any number of labels. Each label is a set of key-value pairs, for example:

```
{
    class:  "rect",
    id:     "Martin",
    x:      10,
    y:      30,
```

```
    width:  40,
    height: 50,
}
```

The only required key a label *has* to have is the "class" key. It will be used by the label tool to determine the appropriate visualization for this label (in our example it will draw a rectangle). You will later see, how you can customize the mapping between class and visualization and how to write your own visualizations.

### 2.2.2 Label type conventions

Sloth provides support for a range of standard shape labels (for example *rect*, *point*, *polygon* etc.). In order for the label tool to correctly visualize these labels, the labels have to follow a convention, which keys represent the *x*- and *y*-coordinates, *width* and *height* and so on.

The following simple geometric classes are supported out of the box, i.e. corresponding visualization items and inserters will be avaible in the default configuration.

#### Point

```
{
    "class": "point",
    "x":     10,
    "y":     20,
}
```

#### Rect

```
{
    "class":   "rect",
    "x":       10,
    "y":       20,
    "width":   20,
    "height":  20,
}
```

#### Polygon

```
{
    "class": "polygon",
    "xn":    "10;20;30",
    "yn":    "20;30;40",
}
```

### 2.2.3 User defined labels

In many cases, it will not be sufficient for your labeling needs to stick to those simple classes. Or, you might want to add further information. Since each label is just a set of key-value pairs, this is easily possible by adding more key-value pairs that carry additional information. For example you can add a key `type` that differentiates point labels to be either the label for the left or the right eye of a face:

```
{
    "class": "point",
    "type":  "left_eye",
    x: 50, y: 40,
},
{
    "class": "point",
    "type":  "right_eye",
    x: 70, y: 40,
}
```

Of course, you can also create new classes:

```
{
    "class": "triangle",
    "x1": 10,
    "y1": 20,
    "x2": 30,
    "y2": 20,
    "x3": 20,
    "y3": 30,
},
{
    "class": "deathstar",
    "x": 678,
    "y": 890,
    "z": 666,
    "range": "very far",
    "last_known_message": "What happens if I press *this* button?"
}
```

You see in the second example, that the label does not necessarily have to name a geometric form of any sort. Neither do the key-value pairs have to denote only coordinates or attributes. It can be anything you like. However, if you create your own classes you will need to tell the label tool in the configuration how to display this label class. See section Configuration on how to do that.

### 2.2.4 Representation is not storage

In the sections above we introduced the labels as sets of key-value pairs with a textual representation. The storage on disk of the labels however can be very different. Sloth does not have *the one* way in which it stores the labels on-disk. The labels could be stored as XML, as binary data or in a textual format. In fact, the labels might not even be stored in a file, but uploaded to a web server. Again, there are some default formats which the label tool can deal with out of the box (among others YAML and JSON, which resemble the textual representation above). However, you are free to define your own loading and saving routines for your labels (see Containers). This allows you for example to support legacy third-party label formats (for example one that comes with a data set) without the need of converting them to JSON first.

## 2.3 First Steps

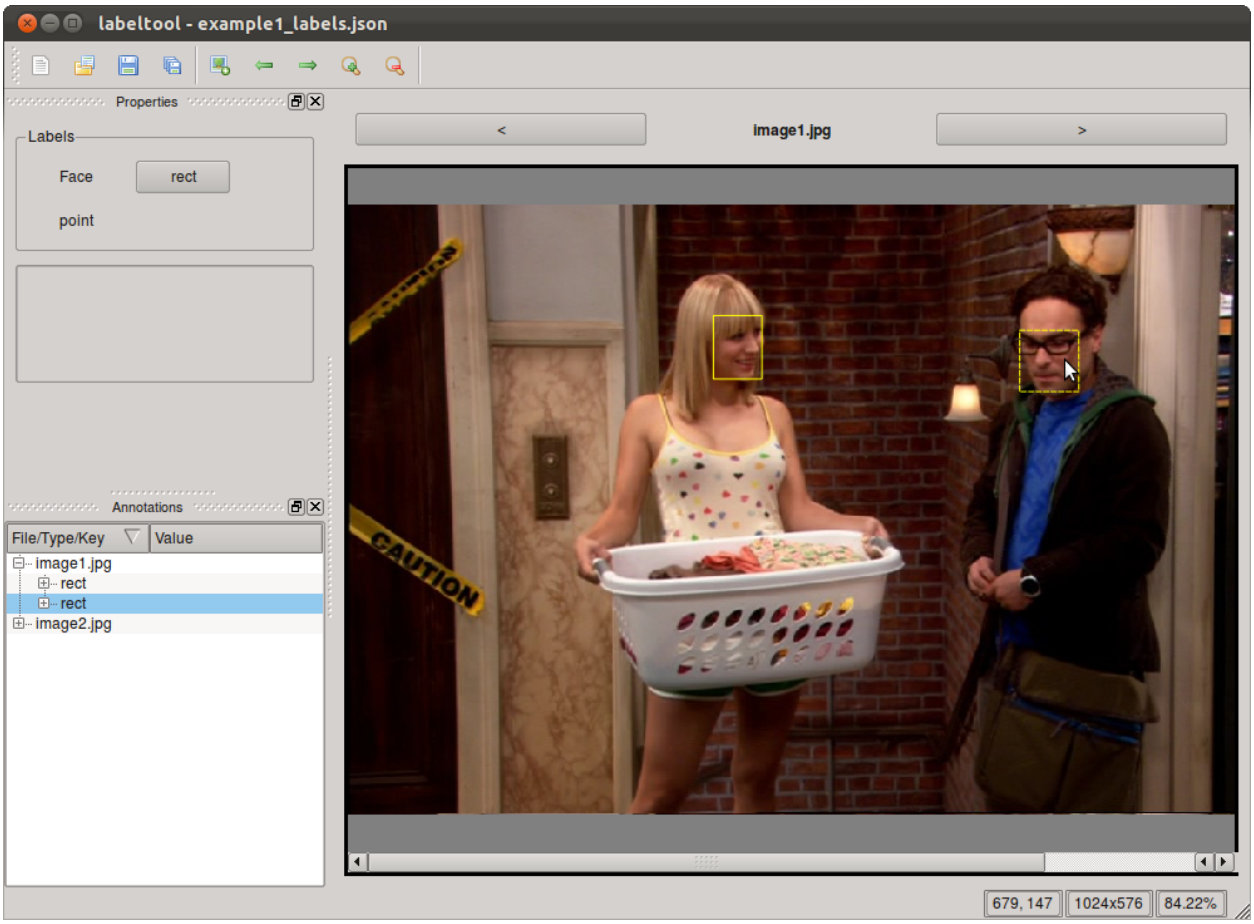
In this section, you will learn with a simple example, how to load labels and write a simple configuration file. The full configuration options will be covered in the next section Configuration.

### 2.3.1 Using the default configuration

The easiest way to start Sloth is by using a supported label format and supported label types only. In that case we just need to start Sloth and supply the label file as parameter on the command line:

```
sloth examples/example1_labels.json
```

`example1_labels.json` comes with Sloth in the example directory for you to try out directly. Once the label file is loaded, Sloth should look somewhat similar to the following image.




Let's take look at the example label file:

```
[
    {
        "class": "image",
        "filename": "image1.jpg",
        "annotations": [
            {
                "class": "rect",
                "height": 60.0,
                "width": 46.0,
                "y": 105.0,
                "x": 346.0
            },
            {
                "class": "rect",
                "height": 58.0,
```

```
                "width": 56.0,
                "y": 119.0,
                "x": 636.0
            }
        ]
    },
    {
        "class": "image",
        "filename": "image2.jpg",
        "annotations": [
            {
                "class": "point",
                "y": 155.0,
                "x": 409.0
            }
        ]
    }
]
```

We have labeled two images with filenames `image1.jpg` and `image2.jpg`, with two rectangles in image 1 and one point in image 2. Since we launched Sloth without a custom configuration, the standard visualizations for `rect` and `point` will be used. Sloth displays two rectangles at the labeled positions in image1, and a point in image2.

### 2.3.2 Adding and editing annotations in the GUI

#### Editing existing annotations

Let's start by editing existing labels. There are several ways in which existing labels can be modified. You first need to select the label which you want to modify. You can do this by clicking on the label, e.g. in our example label file you can click somewhere inside the area of one of the rectangles in image 1. The label outline changes to a dashed line, indicating that is has been selected. Another way to select labels is to press the TAB-key multiple times. This cycles the selection through all labels in the current image. Here too, the currently selected item is indicated by a dashed outline.

Once the label is selected, we can modify its position dragging the item to its new location while holding the left mouse button down. This applies to both rectangle and point labels.

In order to modify the width and height of the rectangle, you can click inside the rectangle with the right mouse button and drag while holding down the right mouse button. This changes the width and height of the rectangle.

We strongly belief that in many cases a labeling task can be carried out more efficiently by using the keyboard instead of the mouse. Therefore, the builtin standard label items can all be modified using the keyboard only.

The position of a label item can be modified with the LEFT, RIGHT, UP and DOWN keys with pixel-accuracy. If you hold down the SHIFT-key, the step size is increased to 5-pixel steps.

The width and height of a rectangle can further be modified by holding down the CTRL-key, and then using the LEFT, RIGHT, UP and DOWN buttons, respectively. Again, holding in addition also the SHIFT key increases the step size to 5 pixels

#### Adding new annotations

For each of the label types, there is a button in the Properties dock (by default on the left of the window). Click on this button to change into *insert*-mode. You will now be able to add new label item by clicking and drawing on the current image.

The insert-mode for a particular label type can also be activated by a hotkey. The standard hotkeys for rectangle labels is `r`, and for point labels `p`.

### 2.3.3 Writing a custom configuration

We already briefly touch the subject of configuration. Sloth can be easily tailored to once labeling needs by using different label types, adding own visualization items and container formats. All of this can be specified in the configuration file. The configuration file is a python module where the module-level variables represent the settings. The most important variable is

- *LABELS*: This defines how sloth will display annotations and how the user can insert new ones.

We start with a quick example:

```
LABELS = (
    {"attributes": {"type":  "rect",
                    "class": "head",
                    "id":    ["Martin", "Mika"]},
     "item":     "sloth.items.RectItem",
     "inserter": "sloth.items.RectItemInserter",
     "text":     "Head"
    },

    {"attributes": {"type":  "point",
                    "class": "left_eye",
                    "id":    ["Martin", "Mika"]},
     "item":     "sloth.items.PointItem",
     "inserter": "sloth.items.PointItemInserter",
     "text":     "Left Eye"
    },

    {"attributes": {"type":  "point",
                    "class": "right_eye",
                    "id":    ["Martin", "Mika"]},
     "item":     "sloth.items.PointItem",
     "inserter": "sloth.items.PointItemInserter",
     "text":     "Right Eye"
    },
)
```

`LABELS` is a tuple/list of dictionaries. Each dictionary describes how one annotation type is (i) inserted, (ii) visualized and (iii) modified. Let's go over the different keys of the dictionary in detail:

- `text`: This is a text that describes the label type, and will be used as label description in the Properties dock.
- `item` specifies which class is responsible for visualizing the annotation. For the first annotation type in our example, the predefined `sloth.items.RectItem` class is used, which will draw a rectangle as given by the coordinates in the annotation. Sloth comes with several predefined visualization classes, such as `sloth.items.RectItem` and `sloth.items.PointItem` (see items for a full list). However, it is also very easy to define your own visualization class (see items).
- `inserter` specifies which class is responsible for creating new annotations based on user input. When the user enters insert-mode with a given label type, the corresponding inserter is captures all user input and takes care of the creation of a new annotation.
- `attributes` has three purposes: 1. It defines which key-values pairs are inserted into a new annotation directly.

This can either be a fixed key-value pair. Or, if the value is a list of items, the user can choose interactively in the Properties dock which one of the values he wants to use for a new label. The current state is then passed to the inserter.

2. It defines how a existing annotations can be edited. Fixed key-value pairs, are not allowed to be edited. If the value for a given key is a list of items, the user can choose interactively between the values for the corresponding key. The annotation is then updated accordingly.
3. It defines how to match an existing annotation to one of the entries in `LABELS`. Sloth uses a soft matching based on the two keys `class` and `type`. It checks each item in `LABELS` starting from the beginning and stops if it finds the first match. An entry matches an annotation if:
   - the values for both the `class` and `type` keys match, or
   - the value for one of keys matches and the other key is not present in either `attributes` or the annotation.

You need to save your custom configuration in a file ending with `.py`. To use it, pass it to Sloth with the `--config` command line parameter:

```
sloth --config myconfig.py examples/example1_labels.json
```

You can now start labeling head locations and eye positions. You'll see that for each depending on the chosen annotation, you can either insert a rectangle (this is internally done by the `RectItemInserter`) or points (using the `PointItemInserter`). For each annotation you can choose an identity between the two supplied options.

There are more possibilities to configure the labels, e.g. defining hotkeys, which we have not touched here. Refer to *LABELS* for the full documentation.

Apart from defining the supported labels in the configuration, other parts of Sloth's behaviour can be configured there as well, e.g. for supporting own label formats with custom containers. See Configuration for the full reference of all configuration options.

### 2.3.4 Next steps

You can now continue by reading about all available configuration options, how to write your own visualization items, custom inserters or custom label containers.

## 2.4 Configuration

The configuration file is a python module where the module-level variables represent the settings.

### 2.4.1 Settings

This is a list of all available settings.

#### LABELS

Default:

```
(
    {
        'attributes': {
            'type':       'rect',
```

```
        },
        'inserter': 'sloth.items.RectItemInserter',
        'item':     'sloth.items.RectItem',
        'hotkey':   'r',
        'text':     'Rectangle',
    },
    {
        'attributes': {
            'type':    'point',
        },
        'inserter': 'sloth.items.PointItemInserter',
        'item':     'sloth.items.PointItem',
        'hotkey':   'p',
        'text':     'Point',
    },
)
```

`LABELS` is a tuple/list of dictionaries. Each dictionary describe how one annotation type is visualized, newly inserted and modified. Let's go over the different keys of the dictionary in detail:

- `text`: This is a text that describes the label type, and will be displayed to the user in the GUI.
- `item` specifies which class is responsible for visualizing the annotation. For the first annotation type we chose to use the predefined `sloth.items.RectItem` class, which will draw a rectangle as given by the coordinates in the annotation. Sloth comes with several predefined visualization classes, such as `sloth.items.RectItem` and `sloth.items.PointItem` (see Items for a full list). However, it is also very easy to define your own visualization class (see *Write your own visualization item*).
- `inserter` specifies which class is responsible for creating new annotations based on user input. When the user enters insert mode with a given label type, the corresponding inserter is instantiated and captures all user input for the creation of a new annotation. The inserter is passed the current state of the button area.
- `attributes` has three functions: 1. It defines how a new annotation can be initialized. Fixed

  key-value pairs are used directly. If the value is a list of items, the user can choose interactively which one of the values he wants to use for a new label. The current state is then passed to the inserter.

  2. It defines how a existing annotations can be edited. Fixed key-value are not allowed to be edited. If the value is a list of items, the user can choose interactively between the values for the corresponding key. The annotation is then updated accordingly.
  3. It defines how to match an existing annotation to one of the entries in `LABELS`. Sloth uses a soft matching based on the two keys `class` and `type`. It checks each item in `LABELS` starting from the beginning and stops if it finds the first match. An entry matches an annotation if:
     - the values for both keys match, or
     - the value for one of keys matches and the other key is not present in either `attributes` or the annotation.

Note that the comma at the end of the first tuple is mandatory. Otherwise the outer tuple will not be recognized as one (it will be only parentheses around an object, which will alone not be translated into a tuple object. This applies similarly to all tuple/list-type settings.

### HOTKEYS

Default:

```
(
    ('PgDown',    lambda lt: lt.gotoNext(),                  'Next image/frame'),
    ('PgUp',      lambda lt: lt.gotoPrevious(),              'Previous image/frame'),
    ('Tab',       lambda lt: lt.selectNextAnnotation(),      'Select next annotation'),
    ('Shift+Tab', lambda lt: lt.selectPreviousAnnotation(),  'Select previous annotation'),
    ('Del',       lambda lt: lt.deleteSelectedAnnotations(), 'Delete selected annotations'),
    ('ESC',       lambda lt: lt.exitInsertMode(),            'Exit insert mode'),
)
```

Defines global keyboard shortcuts. Each hotkey is defined by a tuple with at least 2 entries, where the first entry is the hotkey (sequence), and the second entry is the function that is called. The function should expect a single parameter, the labeltool object. The optional third entry – if present – is expected to be a string describing the action.

#### CONTAINERS

Default:

```
{
 '*.txt':    'annotations.container.SimpleOneLinerTextContainer',
 '*.yaml':   'annotations.container.YamlContainer',
 '*.pickle': 'annotations.container.PickleContainer',
}
```

Defines a mapping of which container should be used for loading a label file matching the given filename pattern. This can of course also be a user defined container. You can also define the class directly (instead of a module path):

```
{
 '*.foo':   MyFooContainer
}
```

#### PLUGINS

Did not think to much about this yet. This is rather for v2.0. Could image to be able to define some kind of plugin that might do some preprocessing on an image, e.g. detect all faces and convert them into labels.

#### SCENE_BACKGROUND

Default:

```
Qt.darkGray
```

Allows to set the scene background to a custom color or pattern. Expects a QBrush. A more complex background could be a regular box pattern which might simplify the exact resizing of annotations that extend over image boundaries:

```
from PyQt4.QtGui import QBrush
from PyQt4.QtCore import Qt
SCENE_BACKGROUND = QBrush(Qt.darkGray, Qt.CrossPattern)
```

### 2.4.2 Extending default values

In the usual case one overrides the default when defining a configuration variable. In order to extend the default configuration and avoid overriding the default values, you can first import the default configuration and then append your custom mappings (remember that the configuration is a python module, therefore you can execute any valid python code):

```
from sloth.conf.default_config import LABELS

MYLABELS = ({
    ...
})

LABELS += MYLABELS
```

## 2.5 Items

Visualization items are reponsible for bringing the labels to the users screen. The visualization in the label tool is object based, i.e. for each label the label tool creates a item object that is responsible for drawing the label.

### 2.5.1 Predefined items

The label tool comes with a few predefined visualization items:

- `items.PointItem`

  Draws a point. Expects the label to have keys `x` and `y` with the coordinates as values.

- `items.RectItem`

  Draws a rectangle. Expects the label to have keys `x`, `y`, `width` and `height`.

- `items.PolygonItem`

  Draws a rectangle. Expects the label to have keys `xn` and `yn`, which are `;`-separated lists of point coordinates.

The predefined items can be used in different ways. If you give the class name in the configuration, the constructor will be called for initializing the item. However, you can also create and instance of the item, configure for example the color, and then use this instance in the configuration. The predefined items have their `__call__` operator overloaded and will function as a factory creating new items similar to the current instance. You can make use of this in the configuration to for example specify the color of the created rectangles, maybe even different kinds for different label types:

```
# this your custom configuration module

RedRectItem = items.RectItem()
RedRectItem.setColor(Qt.Red)
GreenRectItem = items.RectItem()
GreenRectItem.setColor(Qt.Green)

ITEMS = {
    "rect" : RedRectItem,
    "head" : GreenRectItem,
}
```

### 2.5.2 items.RectItem

Usage:

- Can be moved by Left/Right/Up/Down keys. If Shift is pressed, step is increased. If Control is pressed, width and height are modified instead of position.

### 2.5.3 Write your own visualization item

The base class for all visualization item is the BaseItem class. In order to write a new visualization item, you need to subclass this class and implement a few functions.

The easiest way to visualize your label is by using some of the existing Qt graphics items. You can initialize it in the constructor and be done:

```
class MyRectItem(BaseItem):
    def __init__(self, index, data):
        # Call the base class constructor.  This will make the label
        # data available in self.data
        BaseItem.__init__(self, index, data)

        # Create a new rect item and add it as child item.
        # This defines what will be displayed for this label, since the
        # BaseItem base class itself does not display anything.
        x, y, width, height = map(float, (self.data['x'],     self.data['y'],
                                          self.data['width'], self.data['height']))
        self.rect_ = QGraphicsRectItem(x, y, width, height, self)
```

For advanced usage, for example allowing the label to be moved by the mouse, we need to do some more. First, we need to allow the item to be selectable and movable. In the constructor set the graphics items flags to allow interactive modfications of the item:

```
self.setFlags(QGraphicsItem.ItemIsSelectable | \
              QGraphicsItem.ItemIsMovable | \
              QGraphicsItem.ItemSendsGeometryChanges | \
              QGraphicsItem.ItemSendsScenePositionChanges)
```

Then we catch the notifications about item changes by overriding `ìtemChange`. We especially need to inform the model about the modification:

```
def itemChange(self, change, value):
    if change == QGraphicsItem.ItemScenePositionHasChanged:
        self.updateModel()
    return AnnotationGraphicsItem.itemChange(self, change, value)

def updateModel(self):
    rect = QRectF(self.scenePos(), self.rect_.size())
    self.data['x']      = rect.topLeft().x()
    self.data['y']      = rect.topLeft().y()
    self.data['width']  = float(rect.width())
    self.data['height'] = float(rect.height())

    self.index().model().setData(self.index(), QVariant(self.data), DataRole)
```

For even more advanced usage, such as drawing your own shapes, catching keys etc., please consult Qt's QGraphicsItem documentation.

### 2.5.4 Factorize your custom visualization item

The predefined items are implemented in such a way so that they can be used as template to create new, similar items. In order to implement something similar for your own visualization items, you need to overload your classes `__call__` operator and return a new visualization item with all properties cloned that you would like to clone.

Example:

```
class MyRectItem(BaseItem):
    def __init__(self, index, data):
        BaseItem.__init__(self, index, data)
        self.color_ = Qt.Red

    def setColor(self, color):
        self.color_ = color

    def __call__(self, index, data):
        newitem = MyRectItem(index, data)
        newitem.setColor(self.color_)
        return newitem
```

You can see that the `__call__` operator takes the same arguments as the constructor. In its implementation it first creates a new visualization item, and then sets the color to the same as its own before returning the new item.

## 2.6 Inserters

Inserters are used for creating new labels interactively. When the users selects a label type in the button area, the corresponding inserter for the label type (as defined in the configuration).

**Todo**

Write this. It's pretty similar to the items section.

## 2.7 Containers

Annotation containers provide functions for loading and saving labels. You can write custom containers to support specific label formats.

### 2.7.1 Container Interface

A container is expected to implement (at least) these five functions:

**load** (*self*, *filename*)
Loads and returns the annotations in file `filename`.

**save** (*self*, *annotations*, *filename*)
Writes the given annotations to file `filename`.

**filename** (*self*)
Returns the current filename.

**loadImage** (*self*, *filename*)
Loads and returns the image referenced to by filename

**loadFrame** (*self*, *filename*, *frame_number*)
Load the video referenced to by the filename, and return frame `frame_number`.

The container base class `AnnotationContainer` provides default implementations for all five function. It however deferes the parsing and serialization of the labels from/to disk to the to functions

**parseFromFile** (*self*, *filename*)

and

**serializeToFile**(*self*, *filename*, *annotations*)

respectively. If you subclass AnnotationContainer, make sure to provide implementations for those two functions.

### 2.7.2 Default Containers

A few containers are included in sloth. They can be found in the module `sloth.annotations.container`. In the default configuration, these containers are included for their respective default filename patter.

#### JsonContainer

Default pattern: `*.json`

Writes and reads annotations in JSON format (needs the python module `json` to be installed).

#### YamlContainer

Default pattern: `*.yaml`

Writes and reads annotations in YAML format (needs the python module `yaml` to be installed).

#### PickleContainer

Default pattern: `*.pickle`

Writes and reads annotations in pickle format (needs the python module `pickle` or `cPickle` to be installed, `cPickle` is more performant).

#### FileNameListContainer

Default pattern: `*.sloth-init`

A simple container that reads one image filename per line. No annotations are supported. This container can be used for example for initializing a labeling session. After adding labels, another container should be used for saving though, otherwise the labels will be lost. (write support not implemented yet anyway)

#### FeretContainer

Reads annotations in the Feret format (no write support implemented yet). This container is not included in the default configuration.

## 2.8 Examples

### 2.8.1 Adding every nth image to label file

This can be achieved by a combination of `find` and `awk`:

```
find shot01/ -iname "*.png" | sort | awk 'NR%5==1' | xargs sloth appendfiles shot01.json
```

## 2.9 API Reference

**Todo**

Actually document the code, so that this is not just a bunch of method names.

### 2.9.1 Labeltool

The labeltool object is the main object that hold most of the current state of the label tool.

It provides the following API:

### 2.9.2 Containers

This details the default containers that come with sloth.

The following containers are available in `sloth.annotations.container`:

### 2.9.3 Model

This details the model API.

### 2.9.4 Items

This details the default visualization items that come with sloth.

The following items are available in *`sloth.items`*:

### 2.9.5 Inserters

This details the default inserters that come with sloth.

The following inserters are available in *`sloth.items`*:

### 2.9.6 Scene

This details the scene API.

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

## S

## F



## L



## P



## S