Augmentor Documentation

Release 0.2.12

Marcus D. Bloice

Mar 29, 2023

User Guide

1 Main Features 3
1.1 Perspective Skewing 4
1.2 Elastic Distortions 5
1.3 Rotating 6
1.4 Shearing 8
1.5 Cropping 9
1.6 Mirroring 11
1.7 Notes 12

2 Installation 13
2.1 Requirements 13
2.2 Building 13

3 Usage 15
3.1 Getting Started 15
3.1.1 Step 1: Create a New Pipeline 15
3.1.2 Step 2: Add Operations to the Pipeline 15
3.1.3 Step 3: Execute and Sample From the Pipeline 16

4 Examples 17
4.1 Initialising a pipeline 17
4.2 Adding operations to a pipeline 17
4.3 Executing a pipeline 18

5 Extending Augmentor 19
5.1 Step 1: Create a New Operation Subclass 19
5.2 Step 2: Add an Object to the Pipeline Manually 20
5.3 Using non-PIL Image Objects 20

6 Auto Generated Documentation 23
6.1 Module by Module Documentation 23
6.1.1 Documentation of the Pipeline module 23
6.1.2 Documentation of the Operations module 23
6.1.3 Documentation of the ImageUtilities module 23

7 Licence 25
7.1 Augmentor Licence 25

8 Indices and tables 27

Augmentor is a Python package designed to aid the augmentation and artificial generation of image data for machine learning tasks. It is primarily a data augmentation tool, but will also incorporate basic image pre-processing functionality.

Tip: A Julia version of the package is also being actively developed. If you prefer to use Julia, you can find it here.

The documentation is organised as follows:

CHAPTER 1

Main Features

In this section we will describe the main features of Augmentor with example code and output.

Augmentor is software package for image augmentation with an emphasis on providing operations that are typically used in the generation of image data for machine learning problems.

In principle, Augmentor consists of a number of classes for standard image manipulation functions, such as the `Rotate` class or the `Crop` class. You interact and use these classes using a large number of convenience functions, which cover most of the functions you might require when augmenting image datasets for machine learning problems.

Because image augmentation is often a multi-stage procedure, Augmentor uses a **pipeline**-based approach, where **operations** are added sequentially in order to generate a pipeline. Images are then passed through this pipeline, where each operation is applied to the image as it passes through.

Also, Augmentor applies operations to images **stochastically** as they pass through the pipeline, according to a user-defined probability value for each operation.

Therefore every operation has at minimum a probability parameter, which controls how likely the operation will be applied to each image that is seen as the image passes through the pipeline. Take for example a rotate operation, which is defined as follows:

```
rotate(probability=0.5, max_left_rotation=5, max_right_rotation=10)
```

The `probability` parameter controls how often the operation is applied. The `max_left_rotation` and `max_right_rotation` controls the degree by which the image is rotated, **if** the operation is applied. The value, in this case between -5 and 10 degrees, is chosen at random.

Therefore, Augmentor allows you to create an augmentation pipeline, which chains together operations that are applied stochastically, where the parameters of each of these operations are also chosen at random, within a range specified by the user. This means that each time an image is passed through the pipeline, a different image is returned. Depending on the number of operations in the pipeline, and the range of values that each operation has available, a very large amount of new image data can be created in this way.

All functions described in this section are made available by the Pipeline object. To begin using Augmentor, you always create a new Pipeline object by instantiating it with a path to a set of images or image that you wish to augment:

```
>>> import Augmentor
>>> p = Augmentor.Pipeline("/path/to/images")
Initialised with 100 images found in selected directory.
```

You can now add operations to this pipeline using the `p` Pipeline object. For example, to add a rotate operation:

```
>>> p.rotate(probability=1.0, max_left_rotation=5, max_right_rotation=10)
```

All pipeline operations have at least a probability parameter.

To see the status of the current pipeline:

```
>>> p.status()
There are 1 operation(s) in the current pipeline.
Index 0:
    Operation RotateRange (probability: 1):
        Attribute: max_right_rotation (10)
        Attribute: max_left_rotation (-5)
        Attribute: probability (1)

There are 1 image(s) in the source directory.
Dimensions:
    Width: 400 Height: 400
Formats:
    PNG
```

You can remove operations using the `remove_operation(index)` function and the appropriate `index` indicator from above.

Full documentation of all functions and operations can be found in the auto-generated documentation. This guide suffice as a rough guide to the major features of the package, however.

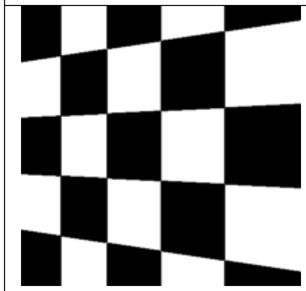
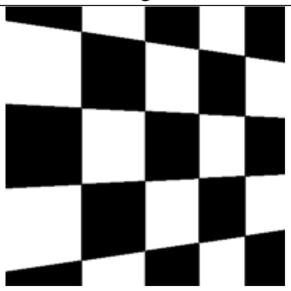
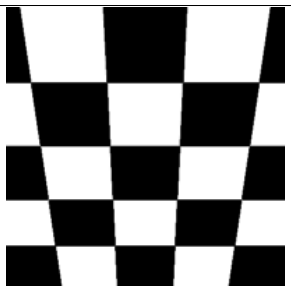
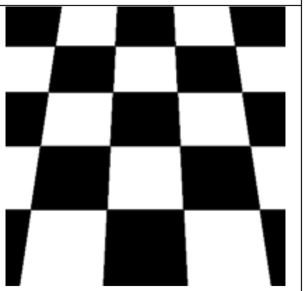
1.1 Perspective Skewing

Perspective skewing involves transforming the image so that it appears that you are looking at the image from a different angle.

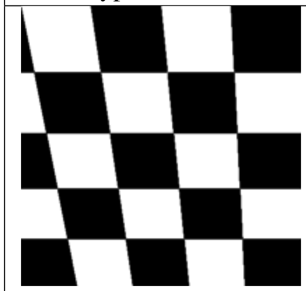
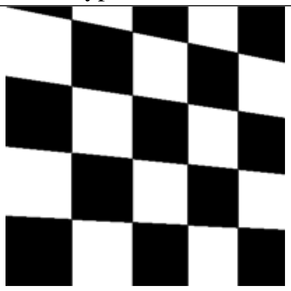
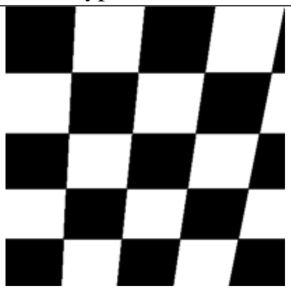
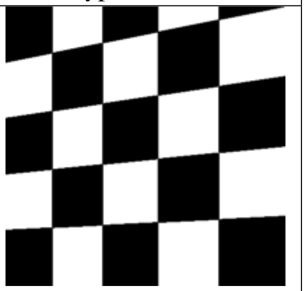
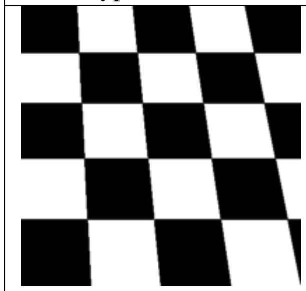
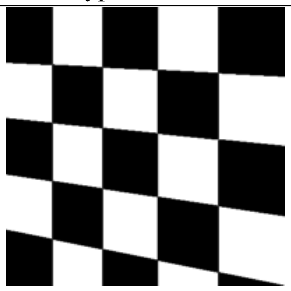
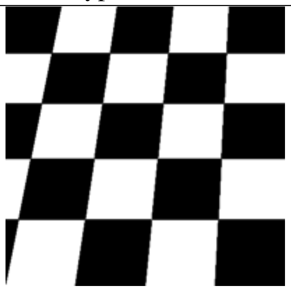
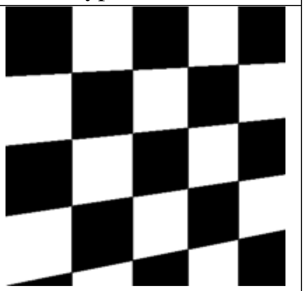
The following main functions are used for skewing:

- `skew_tilt()`
- `skew_left_right()`
- `skew_top_bottom()`
- `skew_corner()`
- `skew()`

To skew or tilt an image either left, right, forwards, or backwards, use the `skew_tilt` function. The image will be skewed by a random amount in the following directions:

Skew Tilt Left	Skew Tilt Right	Skew Tilt Forward	Skew Tilt Backward
			

Or, to skew an image by a random corner, use the `skew_corner()` function. The image will be skewed using one of the following 8 skew types:

Skew Type 0	Skew Type 1	Skew Type 2	Skew Type 3
			
Skew Type 4	Skew Type 5	Skew Type 6	Skew Type 7
			

If you only wish to skew either left or right, use `skew_left_right()`. To skew only forwards or backwards, use `skew_top_bottom()`.

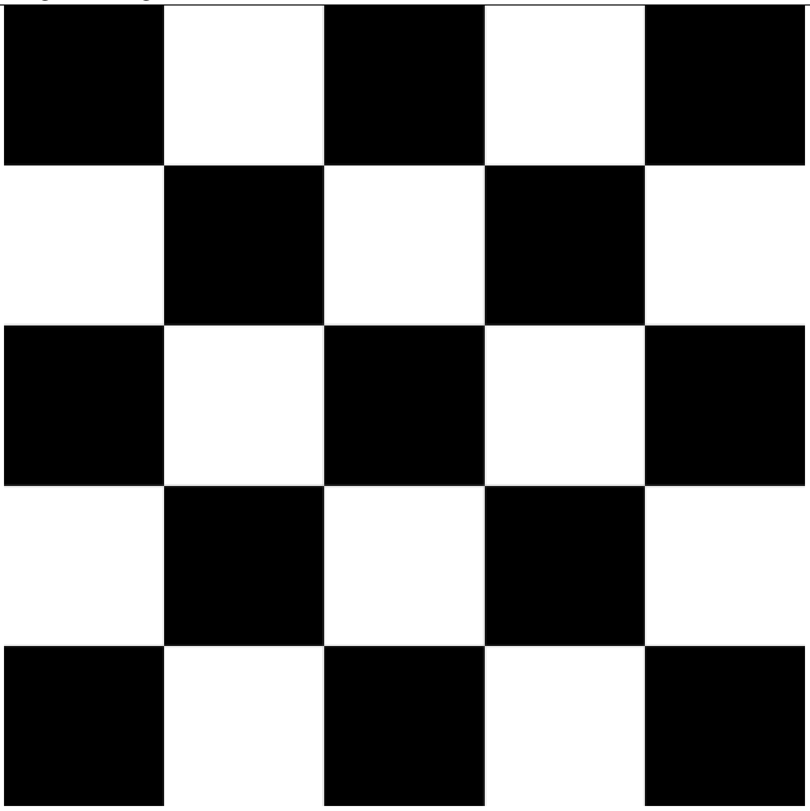
The function `skew()` will skew your image in a random direction of the 12 directions shown above.

1.2 Elastic Distortions

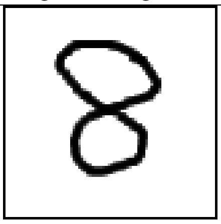
Elastic distortions allow you to make distortions to an image while maintaining the image's aspect ratio.

- `random_distortion()`

Here, we have taken a sample image and generated 50 samples, with a grid size of 16 and a distortion magnitude of 8:

Original Image	Random distortions applied
	

To highlight how this might be useful in a real-world scenario, here is the distort function being applied to a single image of a figure 8.

Original Image	Random distortions applied
	

Realistic new samples can be created using this method.

See the auto-generated documentation for more details regarding this function's parameters.

1.3 Rotating

Rotating can be performed in a number of ways. When rotating by modulo 90, the image is simply rotated and saved. To rotate by arbitrary degrees, then a crop is taken from the centre of the newly rotated image.

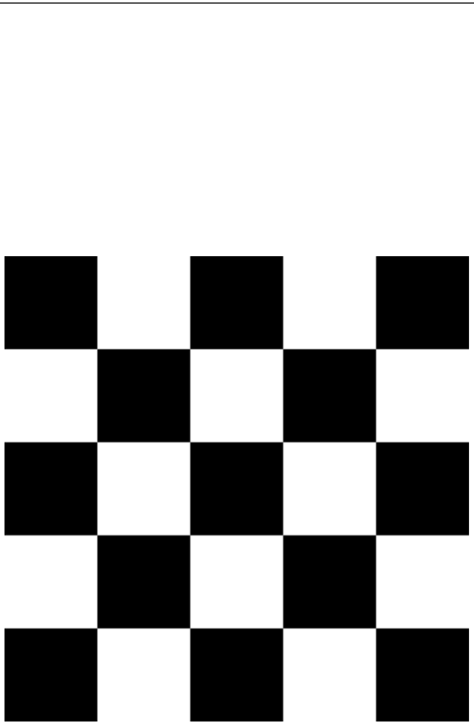
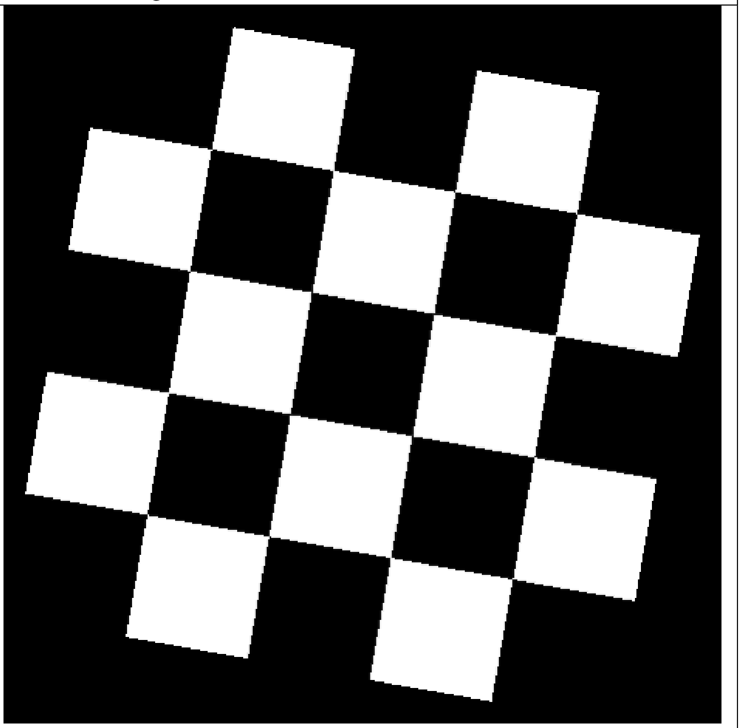
Rotate functions that are available are:

- `rotate()`

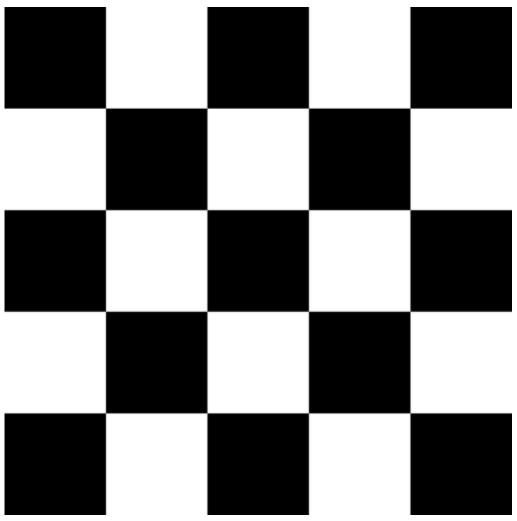
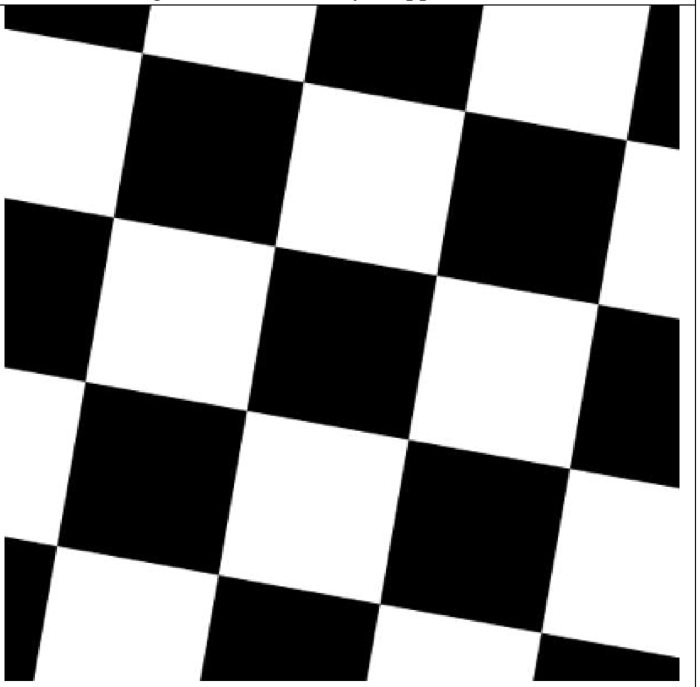
- `rotate90()`
- `rotate180()`
- `rotate270()`
- `rotate_random_90()`

Most of these methods are self-explanatory. The `rotate_random_90()` function will rotate the image by either 90, 180, or 270 degrees.

However, the `rotate()` warrants more discussion and will be described here. When an image is rotated, and it is not a multiple of 90 degrees, the image must either be stretched to accommodate a now larger image, or some of the image must be cut, as demonstrated below:

Original Image	Rotated 10 degrees
	

As can be seen above, an arbitrary, non-modulo 90, rotation will unfortunately result in the image being padded in each corner. To alleviate this, Augmentor's default behaviour is to crop the image and retain the largest crop possible while maintaining the image's aspect ratio:

Original Image	Rotated 10 degrees, automatically cropped
	

This will, of course, result in the image being zoomed in. For smaller rotations of between -5 and 5 degrees, this zoom effect is not particularly drastic.

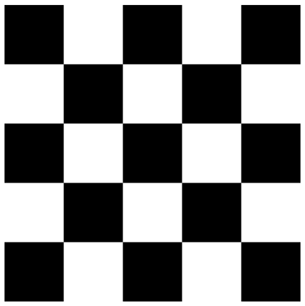
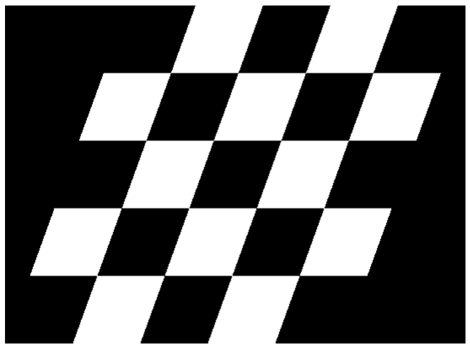
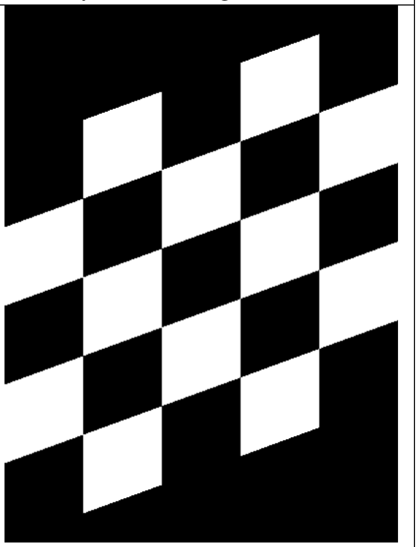
1.4 Shearing

Shearing tilts an image along one of its sides. The can be in the x-axis or y-axis direction.

Functions available for shearing are:

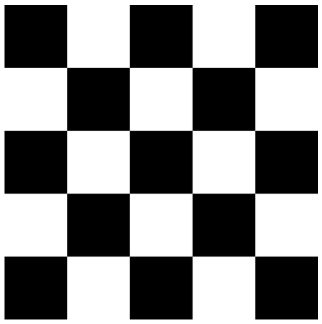
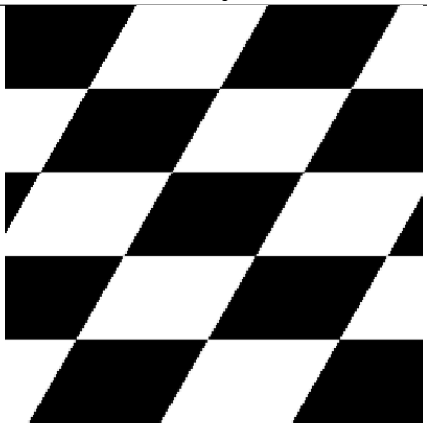
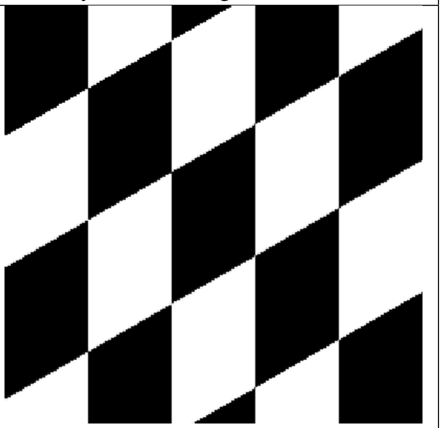
- `shear()`

If you shear in the x or y axis, you will normally get images that look as follows:

Original image	Shear (x-axis) 20 degrees	Shear (y-axis) 20 degrees
		

However, as with rotations, you are left with image that are either larger in size, or are cropped to the original size but contain padding in at the sides of the images.

Augmentor automatically crops the largest area possible before returning the image, as follows:

Original image	Shear (x-axis) 20 degrees	Shear (y-axis) 20 degrees
		

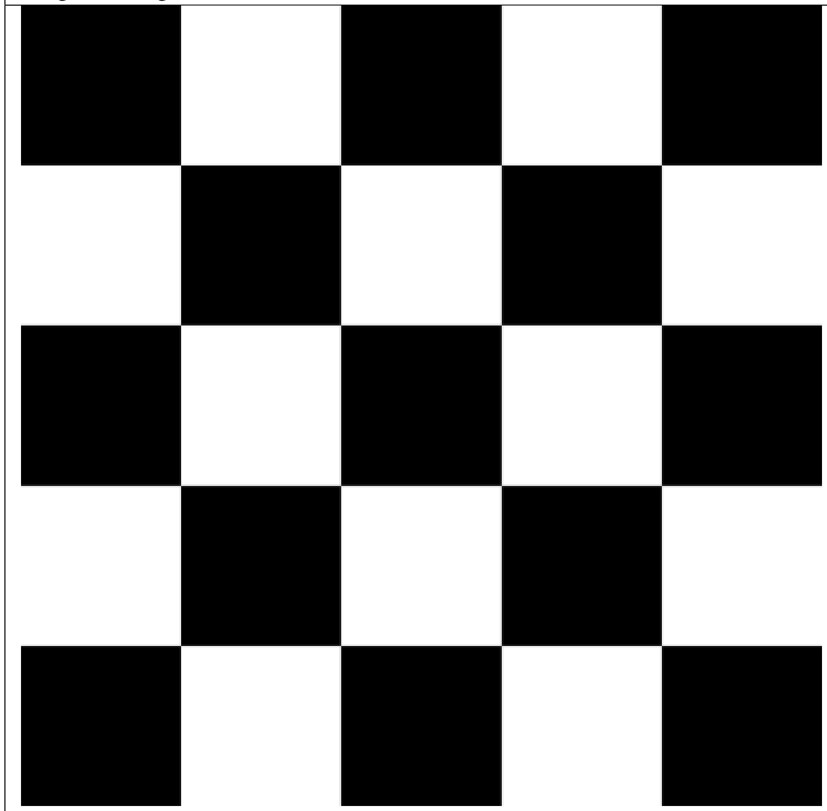
You can shear by random amounts, a fixed amount, in random directions, or in a fixed direction. See the auto-generated documentation for more details.

1.5 Cropping

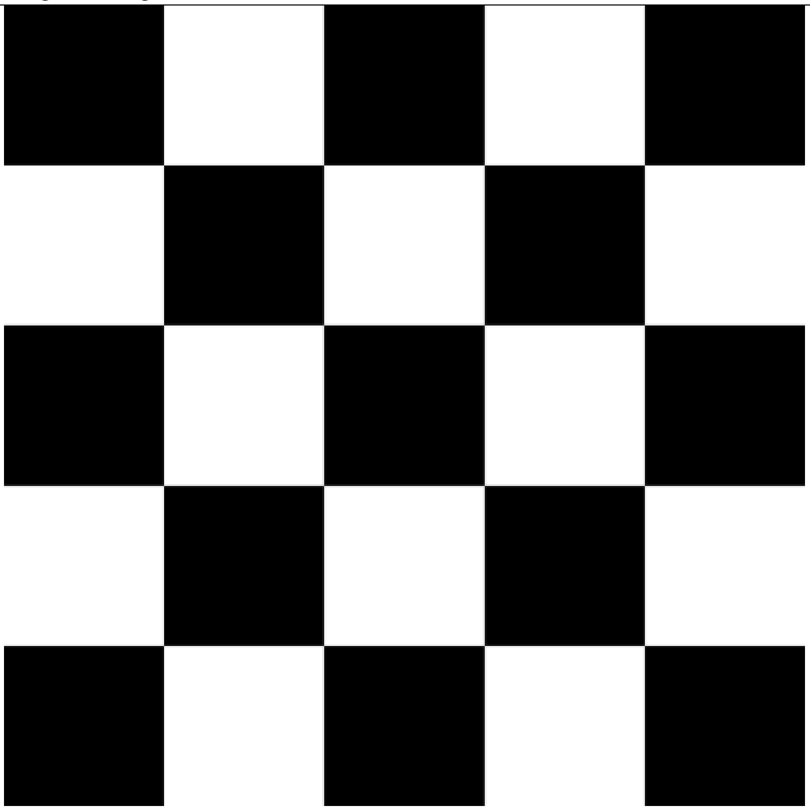
Cropping functions which are available are:

- `crop_centre()`
- `crop_by_size()`
- `crop_random()`

The `crop_random()` function warrants further explanation. Here a region of a size specified by the user is cropped at random from the original image:

Original image	Random crops
	

You could combine this with a resize operation, so that the images returned are the same size as the images of the original, pre-augmented dataset:

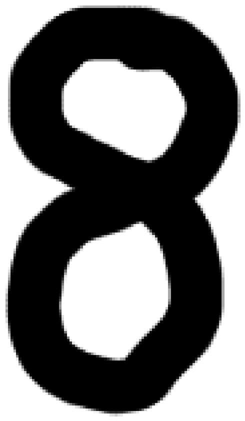
Original image	Random crops + resize operation
	

1.6 Mirroring

The following functions are available for mirroring images (translating them through the x any y axes):

- `flip_left_right()`
- `flip_top_bottom()`
- `flip_random()`

Of these, `flip_random()` can be used in situations where mirroring through both axes may make sense. We may, for example, combine random mirroring, with random distortions, to create new data:

Original image	Random mirroring + random distortions
	

1.7 Notes

Checkerboard image obtained from WikiMedia Commons and is in the public domain. See https://commons.wikimedia.org/wiki/File:Checkerboard_pattern.svg

Skin lesion image obtained from the ISIC Archive:

- Image id: 5436e3adbae478396759f0f1
- Image name: ISIC_0000017.jpg
- Download: https://isic-archive.com:443/api/v1/image/5436e3adbae478396759f0f1/download

See https://isic-archive.com/#images for further details.

CHAPTER 2

Installation

Installation is via `pip`:

```
pip install Augmentor
```

If you have to use `sudo` it is recommended that you use the `-H` flag:

```
sudo -H pip install Augmentor
```

2.1 Requirements

Augmentor requires `Pillow` and `tqdm`. Note that Pillow is a fork of PIL, but both packages cannot exist simultaneously. Uninstall PIL before installing Pillow.

2.2 Building

If you prefer to build the package from source, first clone the repository:

```
git clone https://github.com/mdbloice/Augmentor.git
```

Then enter the `Augmentor` directory and build the package:

```
cd Augmentor
python setup.py install
```

Alternatively you can first run `python setup.py build` followed by `python setup.py install`. This can be useful for debugging.

Attention: If you are compiling from source you may need to compile the dependencies also, including Pillow. On Linux this means having libpng (`libpng-dev`) and zlib (`zlib1g-dev`) installed.

CHAPTER 3

Usage

Here we describe the general usage of Augmentor.

3.1 Getting Started

To use Augmentor, the following general procedure is followed:

1. You instantiate a `Pipeline` object pointing to a directory containing your initial image data set.
2. You define a number of operations to perform on this data set using your `Pipeline` object.
3. You execute these operations by calling the `Pipeline`'s `sample()` method.

We will go through each of these steps in order in the proceeding 3 sub-sections.

3.1.1 Step 1: Create a New Pipeline

Let us first create an empty pipeline. In other words, to begin any augmentation task, you must first initialise a `Pipeline` object, that points to a directory where your original image dataset is stored:

```
>>> import Augmentor
>>> p = Augmentor.Pipeline("/path/to/images")
Initialised with 100 images found in selected directory.
```

The variable `p` now contains a `Pipeline` object, and has been initialised with a list of images found in the source directory.

3.1.2 Step 2: Add Operations to the Pipeline

Once you have created a `Pipeline`, `p`, we can begin by adding operations to `p`. For example, we shall begin by adding a `rotate()` operation:

```
>>> p.rotate(probability=0.7, max_left_rotation=10, max_right_rotation=10)
```

In this case, we have added a `rotate()` operation, that will execute with a probability of 70%, and have defined the maximum range by which an image will be rotated from between -10 and 10 degrees.

Next, we add a further operation, in this case a `zoom()` operation:

```
>>> p.zoom(probability=0.3, min_factor=1.1, max_factor=1.6)
```

This time, we have specified that we wish the operation to be applied with a probability of 30%, while the scale should be randomly selected from between 1.1 and 1.6

3.1.3 Step 3: Execute and Sample From the Pipeline

Once you have added the operations that you require, you can generate new, augmented data by using the `sample()` function and specify the number of images you require, in this case 10,000:

```
>>> p.sample(10000)
```

A progress bar will appear providing a number of metrics while your samples are generated. Newly generated, augmented images will by default be saved into an directory named **output**, relative to the directory which contains your initial image data set.

Hint: A full list of operations can be found in the `Operations` module documentation.

CHAPTER 4

Examples

A number of typical usage scenarios are described here.

Note: A full list of operations can be found in the `Operations` module documentation.

4.1 Initialising a pipeline

```python
import Augmentor

path_to_data = "/home/user/images/dataset1/"

# Create a pipeline
p = Augmentor.Pipeline(path_to_data)
```

4.2 Adding operations to a pipeline

```python
# Add some operations to an existing pipeline.

# First, we add a horizontal flip operation to the pipeline:
p.flip_left_right(probability=0.4)

# Now we add a vertical flip operation to the pipeline:
p.flip_top_bottom(probability=0.8)

# Add a rotate90 operation to the pipeline:
p.rotate90(probability=0.1)
```

4.3 Executing a pipeline

```
# Here we sample 100,000 images from the pipeline.

# It is often useful to use scientific notation for specify
# large numbers with trailing zeros.
num_of_samples = int(1e5)

# Now we can sample from the pipeline:
p.sample(num_of_samples)
```

CHAPTER 5

Extending Augmentor

Extending Augmentor to add new functionality is quite simple, and is performed in two steps:

1) Create a custom class which subclasses from the `Operation` base class, and
2) Add an object of your new class to the pipeline using the `add_operation()` function.

This allows you to add custom functionality and extend Augmentor at run-time. Of course, if you have written an operation that may be of benefit to the community, you can make a pull request on the GitHub repository.

The following sections describe extending Augmentor in two steps. Step 1 involves creating a new `Operation` subclass, and step 2 involves using an object of your new custom operation in a pipeline.

5.1 Step 1: Create a New Operation Subclass

To create a custom operation and extend Augmentor:

1) You create a new class that inherits from the `Operation` base class.
2) You must overload the `perform_operation()` method belonging to the superclass.
3) You must call the superclass's `__init__()` constructor.
4) You must return an object of type `PIL.Image`.

For example, to add a new operation called `FoldImage`, you would write this code:

```
# Create your new operation by inheriting from the Operation superclass:
class FoldImage(Operation):
    # Here you can accept as many custom parameters as required:
    def __init__(self, probability, num_of_folds):
        # Call the superclass's constructor (meaning you must
        # supply a probability value):
        Operation.__init__(self, probability)
        # Set your custom operation's member variables here as required:
        self.num_of_folds = num_of_folds
```

(continues on next page)

(continued from previous page)

```
    # Your class must implement the perform_operation method:
    def perform_operation(self, image):
        # Start of code to perform custom image operation.
        for fold in range(self.num_of_folds):
            pass
        # End of code to perform custom image operation.

        # Return the image so that it can further processed in the pipeline:
        return image
```

You have seen that you need to implement the `perform_operation()` function and you must call the superclass's constructor which requires a `probability` value to be set. Ensure you return a PIL Image as a return value.

If you wish to make these changes permanent, place your code in the `Operations` **module**.

Hint: You can also overload the superclass's `__str__()` function to return a custom string for the object's description text. This is useful for some methods that display information about the operation, such as the `status()` method.

5.2 Step 2: Add an Object to the Pipeline Manually

Once you have a new operation which is of type `Operation`, you can add an object of you new operation to an existing pipeline.

```
# Instantiate a new object of your custom operation
fold = Fold(probability = 0.75, num_of_folds = 4)

# Add this to the current pipeline
p.add_operation(fold)

# Executed the pipeline as normal, and your custom operation will be executed
p.sample(1000)
```

As you can see, adding custom operations at run-time is possible by subclassing the `Operation` class and adding an object of this class to the pipeline manually using the `add_operation()` function.

5.3 Using non-PIL Image Objects

Images can be converted to their raw formats for custom operations, for example by using NumPy:

```
import numpy

# Custom class declaration

def perform_operation(image):

    image_array = numpy.array(image).astype('uint8')

    # Perform your custom operations here
```

(continues on next page)

(continued from previous page)

```
    image = PIL.Image.fromarray(image_array)

    return image
```

CHAPTER 6

Auto Generated Documentation

6.1 Module by Module Documentation

6.1.1 Documentation of the Pipeline module

6.1.2 Documentation of the Operations module

6.1.3 Documentation of the ImageUtilities module

CHAPTER 7

Licence

7.1 Augmentor Licence

The Augmentor package is licenced under the terms of the MIT licence.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search