imageio Documentation

Release 2.10.5

imageio contributors

Nov 17, 2021

CONTENTS

Imageio is a Python library that provides an easy interface to read and write a wide range of image data, including animated images, volumetric data, and scientific formats. It is cross-platform, runs on Python 3.5+, and is easy to install.

Main website: https://imageio.readthedocs.io/

Contents:

CHAPTER

ONE

GETTING STARTED

1.1 Installing imageio

Imageio is written in pure Python, so installation is easy. Imageio works on Python 3.5+. It also works on Pypy. Imageio depends on Numpy and Pillow. For some formats, imageio needs additional libraries/executables (e.g. ffmpeg), which imageio helps you to download/install.

To install imageio, use one of the following methods:

- If you are in a conda env: `conda install -c conda-forge imageio`
- If you have pip: `pip install imageio`
- Good old `python setup.py install`

After installation, checkout the *examples* and *user api*.

Still running Python 2.7? Read *here*.

1.2 Imageio Usage Examples

Some of these examples use Visvis to visualize the image data, but one can also use Matplotlib to show the images.

Imageio provides a range of *example images*, which can be used by using a URI like `'imageio:chelsea.png'`. The images are automatically downloaded if not already present on your system. Therefore most examples below should just work.

1.2.1 Read an image of a cat

Probably the most important thing you'll ever need.

```
import imageio as iio

im = iio.imread('imageio:chelsea.png')
print(im.shape)
```

If the image is a GIF:

```
import imageio as iio

im = iio.get_reader('cat.gif')
```

(continues on next page)

(continued from previous page)

```
for frame in im:
    print(frame.shape)  # Each frame is a numpy matrix
```

If the GIF is stored in memory:

```
import imageio as iio

im = iio.get_reader(image_bytes, '.gif')
```

1.2.2 Read from fancy sources

Imageio can read from filenames, file objects, http, zipfiles and bytes.

```
import imageio as iio
import visvis as vv

im = iio.imread('http://upload.wikimedia.org/wikipedia/commons/d/de/Wikipedia_Logo_1.0.
↪png')
vv.imshow(im)
```

Note: reading from HTTP and zipfiles works for many formats including png and jpeg, but may not work for all formats (some plugins "seek" the file object, which HTTP/zip streams do not support). In such a case one can download/extract the file first. For HTTP one can use something like `imageio.imread(imageio.core.urlopen(url).read(), '.gif')`.

1.2.3 Read all Images in a Folder

One common scenario is that you want to read all images in a folder, e.g. for a scientific analysis, or because these are all your training examples. Assuming the folder only contains image files, you can read it using a snippet like

```
import imageio as iio
from pathlib import Path

images = list()
for file in Path("path/to/folder").iterdir():
    im = iio.imread(file)
    images.append(im)
```

1.2.4 Iterate over frames in a movie

```
import imageio as iio

reader = iio.get_reader('imageio:cockatoo.mp4')
for i, im in enumerate(reader):
    print('Mean of frame %i is %1.1f' % (i, im.mean()))
```

1.2.5 Grab screenshot or image from the clipboard

(Screenshots are supported on Windows and OS X, clipboard on Windows only.)

```
import imageio as iio

im_screen = iio.imread('<screen>')
im_clipboard = iio.imread('<clipboard>')
```

1.2.6 Grab frames from your webcam

Use the special <video0> uri to read frames from your webcam (via the ffmpeg plugin). You can replace the zero with another index in case you have multiple cameras attached. You need to `pip install imageio-ffmpeg` in order to use this plugin.

```
import imageio as iio
import visvis as vv

reader = iio.get_reader('<video0>')
t = vv.imshow(reader.get_next_data(), clim=(0, 255))
for im in reader:
    vv.processEvents()
    t.SetData(im)
```

1.2.7 Convert a movie

Here we take a movie and convert it to gray colors. Of course, you can apply any kind of (image) processing to the image here ... You need to `pip install imageio-ffmpeg` in order to use the ffmpeg plugin.

```
import imageio as iio

reader = iio.get_reader('imageio:cockatoo.mp4')
fps = reader.get_meta_data()['fps']

writer = iio.get_writer('~/cockatoo_gray.mp4', fps=fps)

for im in reader:
    writer.append_data(im[:, :, 1])
writer.close()
```

1.2.8 Read medical data (DICOM)

```
import imageio as iio
dirname = 'path/to/dicom/files'

# Read as loose images
ims = iio.mimread(dirname, 'DICOM')
# Read as volume
vol = iio.volread(dirname, 'DICOM')
```

(continues on next page)

(continued from previous page)

```
# Read multiple volumes (multiple DICOM series)
vols = iio.mvolread(dirname, 'DICOM')
```

1.2.9 Volume data

```
import imageio as iio
import visvis as vv

vol = iio.volread('imageio:stent.npz')
vv.volshow(vol)
```

1.2.10 Writing videos with FFMPEG and vaapi

Using vaapi (on Linux only) (intel only?) can help free up resources on your laptop while you are encoding videos. One notable difference between vaapi and x264 is that vaapi doesn't support the color format yuv420p.

Note, you will need ffmpeg compiled with vaapi for this to work.

```
import imageio as iio
import numpy as np

# All images must be of the same size
image1 = np.stack([imageio.imread('imageio:camera.png')] * 3, 2)
image2 = iio.imread('imageio:astronaut.png')
image3 = iio.imread('imageio:immunohistochemistry.png')

w = iio.get_writer('my_video.mp4', format='FFMPEG', mode='I', fps=1,
                        codec='h264_vaapi',
                        output_params=['-vaapi_device',
                                       '/dev/dri/renderD128',
                                       '-vf',
                                       'format=gray|nv12,hwupload'],
                        pixelformat='vaapi_vld')
w.append_data(image1)
w.append_data(image2)
w.append_data(image3)
w.close()
```

A little bit of explanation:

- `output_params`
  - `vaapi_device` speficifies the encoding device that will be used.
  - `vf` and `format` tell ffmpeg that it must upload to the dedicated hardware. Since vaapi only supports a subset of color formats, we ensure that the video is in either gray or nv12 before uploading it. The `or` operation is achieved with `|`.
- `pixelformat`: set to `'vaapi_vld'` to avoid a warning in ffmpeg.
- `codec`: the code you wish to use to encode the video. Make sure your hardware supports the chosen codec. If your hardware supports h265, you may be able to encode using `'hevc_vaapi'`

1.2.11 Writing to Bytes (Encoding)

You can convert ndimages into byte strings. For this, you have to explicitly specify the desired format, as a byte string doesn't carry any information about the format or color space to use. Since backends differ in the way this information should be provided, you also have to explicitly specify the backend (plugin) to use. Note that, if the backend supports writing to file-like objects, the entire process will happen without touching your file-system. If you, for example, want to write with pillow, you can use:

```
from imageio import v3 as iio

# load an example image
img = iio.imread('imageio:astronaut.png')

# png-encoded bytes string
# Note: defaults to RGB color space
png_encoded = iio.imwrite("<bytes>", img, plugin="pillow", format="PNG")

# jpg-encoded bytes string
# Note: defaults to RGB color space
jpg_encoded = iio.imwrite("<bytes>", img, plugin="pillow", format="JPEG")

# RGBA bytes string
# Important Note: omitting mode="RGBA" would result in a corrupted byte string
img = iio.imread('imageio:astronaut.png', mode="RGBA")
jpg_encoded = iio.imwrite("<bytes>", img, plugin="pillow", format="JPEG", mode="RGBA")
```

1.2.12 Writing to BytesIO

Similar to writing to byte strings, you can also write to BytesIO directly.

```
from imageio import v3 as iio
import io

# load an example image
img = iio.imread('imageio:astronaut.png')

# write as PNG
output = io.BytesIO()
iio.imwrite(output, img, plugin="pillow", format="PNG")

# write as JPG
output = io.BytesIO()
iio.imwrite(output, img, plugin="pillow", format="JPEG")
```

1.2.13 Optimizing a GIF using pygifsicle

When creating a GIF using imageio the resulting images can get quite heavy, as the created GIF is not optimized. This can be useful when the elaboration process for the GIF is not finished yet (for instance if some elaboration on specific frames stills need to happen), but it can be an issue when the process is finished and the GIF is unexpectedly big.

GIF files can be compressed in several ways, the most common one method (the one used here) is saving just the differences between the following frames. In this example, we apply the described method to a given GIF *my_gif* using pygifsicle, a porting of the general-purpose GIF editing command-line library gifsicle. To install pygifsicle and gifsicle, read the setup on the project page: it boils down to installing the package using pip and following the console instructions:

```
pip install pygifsicle
```

Now, let's start by creating a gif using imageio:

```
import imageio as iio
import matplotlib.pyplot as plt

n = 100
gif_path = "test.gif"
frames_path = "{i}.jpg"

n = 100
plt.figure(figsize=(4,4))
for i, x in enumerate(range(n)):
    plt.scatter(x/n, x/n)
    plt.xlim(0, 1)
    plt.ylim(0, 1)
    plt.savefig("{i}.jpg".format(i=i))

with iio.get_writer(gif_path, mode='I') as writer:
    for i in range(n):
        writer.append_data(iio.imread(frames_path.format(i=i)))
```

This way we obtain a 2.5MB gif.

We now want to compress the created GIF. We can either overwrite the initial one or create a new optimized one: We start by importing the library method:

```
from pygifsicle import optimize

optimize(gif_path, "optimized.gif") # For creating a new one
optimize(gif_path) # For overwriting the original one
```

The new optimized GIF now weights 870KB, almost 3 times less.

Putting everything together:

```
import imageio as iio
import matplotlib.pyplot as plt
from pygifsicle import optimize

n = 100
gif_path = "test.gif"
```

(continues on next page)

(continued from previous page)

```
frames_path = "{i}.jpg"

n = 100
plt.figure(figsize=(4,4))
for i, x in enumerate(range(n)):
    plt.scatter(x/n, x/n)
    plt.xlim(0, 1)
    plt.ylim(0, 1)
    plt.savefig("{i}.jpg".format(i=i))

with iio.get_writer(gif_path, mode='I') as writer:
    for i in range(n):
        writer.append_data(iio.imread(frames_path.format(i=i)))

optimize(gif_path)
```

1.2.14 Reading Images from ZIP archives

Note: In the future, this syntax will change to better match the URI standard by using fragments. The updated syntax will be "Path/to/file.zip#path/inside/zip/to/image.png".

```
import imageio as iio

image = iio.imread("Path/to/file.zip/path/inside/zip/to/image.png")
```

1.2.15 Reading Multiple Files from a ZIP archive

Assuming there is only image files in the ZIP archive, you can iterate over them with a simple script like the one below.

```
import os
from zipfile import ZipFile
import imageio as iio

images = list()
with ZipFile("imageio.zip") as zf:
    for name in zf.namelist():
        im = iio.imread(name)
        images.append(im)
```

1.3 ImageResources

Reading images isn't always limited to simply loading a file from a local disk. Maybe you are writing a web application and your want to read images from HTTP, or your images are already in memory as a BytesIO object. Maybe you are doing machine learning and decided that it is smart to compress your images inside a ZIP file to reduce the IO bottleneck. All these are examples of different places where image data is stored (aka. resources).

ImageIO supports reading (and writing where applicable) for all of the above examples and more. To stay organized, we group all these sources/resources together and call them `ImageResource`. Often ImageResources are expressed as URIs though sometimes (e.g., in the case of byte streams) they can be python objects, too.

Here, you can find the documentation on what kind of ImageResources ImageIO currently supports together with documentation and an example on how to read/write them.

1.3.1 Files

Arguably the most common type of resource. You specify it using the path to the file, e.g.

```
img = iio.imread("path/to/my/image.jpg")  # relative path
img = iio.imread("/path/to/my/image.jpg")  # absolute path on Linux
img = iio.imread("C:\\path\\to\\my\\image.jpg")  # absolute path on Windows
```

Notice that this is a convenience shorthand (since it is so common). Alternatively, you can use the full URI to the resource on your disk

```
img = iio.imread("file://path/to/my/image.jpg")
img = iio.imread("file:///path/to/my/image.jpg")
img = iio.imread("file://C:\\path\\to\\my\\image.jpg")
```

1.3.2 Byte Streams

ImageIO can directly handle (binary) file objects. This includes `BytesIO` objects (and subclasses thereof) as well as generic objects that implement `close` and a `read` and/or `write` function. Simply pass them into ImageIO the same way you would pass a URI:

```
file_handle = open("some/image.jpg", "rb")
img = iio.imread(file_handle)
```

1.3.3 Standard Images

Standard images are a curated dataset of pictures in different formats that you can use to test your pipelines. You can access them via the `imageio` scheme

```
img = iio.imread("imageio://chelsea.png")
```

A list of all currently available standard images can be found in the section on *Standard Images*.

1.3.4 Web Servers (http/https)

Note: This is primarily intended for publically available ImageResources. If your server requires authentication, you will have to download the ImageResource yourself before handing it over to ImageIO.

Reading http and https is provided directly by ImageIO. This means that ImageIO takes care of requesting and (if necessary) downloading the image and then hands the image to a compatible backend to do the reading itself. It works with any backend. If the backend supports file objects directly, this processes will happen purely in memory.

You can read from public web servers using a URL string

```
img = iio.imread("https://my-domain.com/path/to/some/image.gif")
```

1.3.5 File Servers (ftp/ftps)

Note: This is primarily intended for publically available ImageResources. If your server requires authentication, you will have to download the ImageResource yourself before handing it over to ImageIO.

Reading ftp and ftps is provided directly by ImageIO following the same logic as reading from web servers:

```
img = iio.imread("ftp://my-domain.com/path/to/some/image.gif")
```

1.3.6 Webcam

Note: To access your webcam you will need to have the ffmpeg backend installed:

```
pip install imageio[ffmpeg]
```

With ImageIO you can directly read images (frame-by-frame) from a webcam that is connected to the computer. To do this, you can use the special target:

```
img = iio.imread("<video0>")
```

If you have multiple video sources, you can replace the `0` with the respective number, e.g:

```
img = iio.imread("<video2>")
```

If you need many frames, e.g., because you want to process a stream, it is often much more performant to open the webcam once, keep it open, and read frames on-demand. You can find an example of this in the *list of examples*.

1.3.7 Screenshots

Note: Taking screenshots are only supported on Windows and Mac.

ImageIO gives you basic support for taking screenshots via the special target `screen`:

```
img = iio.imread("<screen>")
```

1.3.8 Clipboard

Note: reading from clipboard is only supported on Windows.

ImageIO gives you basic support for reading from your main clipboard via the special target `clipboard`:

```
img = iio.imread("<clipboard>")
```

1.3.9 ZIP Archives

You can directly read ImageResources from within ZIP archives without extracting them. For this purpose ZIP archives are treated as normal folders; however, nested zip archives are not supported:

```
img = iio.imread("path/to/file.zip/abspath/inside/zipfile/to/image.png")
```

Note that in a future version of ImageIO the syntax for reading ZIP archives will be updated to use fragments, i.e., the path inside the zip file will become a URI fragment.

1.4 Bird's eye view on ImageIO

The aim of this page is to give you a high level overview of how ImageIO works under the hood. We think this is useful for two reasons:

1. If something doesn't work as it should, you need to know where to search for causes. The overview on this page aims to help you in this regard by giving you an idea of how things work, and - hence - where things may go sideways.
2. If you do find a bug and decide to report it, this page helps us establish some joint vocabulary so that we can quickly get onto the same page, figure out what's broken, and how to fix it.

1.4.1 Terminology

You can think of ImageIO in three parts that work in sequence in order to load an image.

ImageIO Core The user-facing APIs (legacy + v3) and a plugin manager. You send requests to iio.core and it uses a set of plugins (see below) to figure out which backend (see below) to use to fulfill your request. It does so by (intelligently) searching a matching plugin, or by sending the request to the plugin you specified explicitly.

Plugin A backend-facing adapter/wrapper that responds to a request from iio.core. It can convert a request that comes from iio.core into a sequence of instructions for the backend that fullfill the request (eg., read/write/iter). A plugin is also aware if its backend is or isn't installed and handles this case appropriately, e.g., it deactivates itself if the backend isn't present.

Backend Library A (potentially 3d party) library that can read and/or write images or image-like objects (videos, etc.). Every backend is optional, so it is up to you to decide which backends to install depending on your needs. Examples for backends are pillow, tifffile, or ffmpeg.

ImageResource A blob of data that contains image data. Typically, this is a file on your drive that is read by ImageIO. However, other sources such as HTTP or file objects are possible, too.

1.4.2 Issues

In this repo, we maintain ImageIO Core as well as all the plugins. If you find a bug or have a problem with either the core or any of the plugins, please open a new issue here.

If you find a bug or have problems with a specific backend, e.g. reading is very slow, please file an issue upstream (the place where the backend is maintained). When in doubt, you can always ask us, and we will redirect you appropriately.

1.4.3 New Plugins

If you end up writing a new plugin, we very much welcome you to contribute it back to us so that we can offer as expansive a list of backends and supported formats as possible. In return, we help you maintain the plugin - note: a plugin is typically different from the backend - and make sure that changes to ImageIO don't break your plugin. This we can only guarantee if it is part of the codebase, because we can (a) write unit tests for it and (b) update your plugin to take advantage of new features. That said, we generally try to be backward-compatible whenever possible.

The backend itself lives elsewhere, usually in a different repository. Not vendoring backends and storing a copy here keeps things lightweight yet powerful. We can, first of all, directly access any new updates or features that a backend may introduce. Second, we avoid forcing users that won't use a specific backend to go through its, potentially complicated, installation process.

1.5 Imageio Standard Images

Imageio provides a number of standard images. These include classic 2D images, as well as animated and volumetric images. To the best of our knowledge, all the listed images are in public domain.

The image names can be loaded by using a special URI, e.g. `imread('imageio:astronaut.png')`. The images are automatically downloaded (and cached in your appdata directory).

- chelsea.bsdf: The chelsea.png in a BSDF file(for testing)
- newtonscradle.gif: Animated GIF of a newton's cradle
- bricks.jpg: A (repeatable) texture of stone bricks
- meadow_cube.jpg: A cubemap image of a meadow, e.g. to render a skybox.

- wood.jpg: A (repeatable) texture of wooden planks
- cockatoo.mp4: Video file of a cockatoo
- stent.npz: Volumetric image showing a stented abdominal aorta
- astronaut.png: Image of the astronaut Eileen Collins
- camera.png: A grayscale image of a photographer
- checkerboard.png: Black and white image of a chekerboard
- chelsea.png: Image of Stefan's cat
- clock.png: Photo of a clock with motion blur (Stefan van der Walt)
- coffee.png: Image of a cup of coffee (Rachel Michetti)
- coins.png: Image showing greek coins from Pompeii
- horse.png: Image showing the silhouette of a horse (Andreas Preuss)
- hubble_deep_field.png: Photograph taken by Hubble telescope (NASA)
- immunohistochemistry.png: Immunohistochemical (IHC) staining
- moon.png: Image showing a portion of the surface of the moon
- page.png: A scanned page of text
- text.png: A photograph of handdrawn text
- wikkie.png: Image of Almar's cat
- chelsea.zip: The chelsea.png in a zipfile (for testing)

1.6 Imageio command line scripts

This page lists the command line scripts provided by imageio. To see all options for a script, execute it with the `--help` option, e.g. `imageio_download_bin --help`.

- `imageio_download_bin`: Download binary dependencies for imageio plugins to the users application data directory. This script accepts the parameter `--package-dir` which will download the binaries to the directory where imageio is installed. This option is useful when freezing an application with imageio. It is supported out-of-the-box by PyInstaller version>=3.2.2.
- `imageio_remove_bin`: Remove binary dependencies of imageio plugins from all directories managed by imageio. This script is useful when there is a corrupt binary or when the user prefers the system binary over the binary provided by imageio.

1.7 Imageio environment variables

This page lists the environment variables that imageio uses. You can set these to control some of imageio's behavior. Each operating system has its own way for setting environment variables, but to set a variable for the current Python process use `os.environ['IMAGEIO_VAR_NAME'] = 'value'`.

- `IMAGEIO_NO_INTERNET`: If this value is "1", "yes", or "true" (case insensitive), makes imageio not use the internet connection to retrieve files (like libraries or sample data). Some plugins (e.g. freeimage and ffmpeg) will try to use the system version in this case.

- `IMAGEIO_FFMPEG_EXE`: Set the path to the ffmpeg executable. Set to simply "ffmpeg" to use your system ffmpeg executable.
- `IMAGEIO_FREEIMAGE_LIB`: Set the path to the freeimage library. If not given, will prompt user to download the freeimage library.
- `IMAGEIO_FORMAT_ORDER`: Determine format preference. E.g. setting this to `"TIFF, -FI"` will prefer the FreeImage plugin over the Pillow plugin, but still prefer TIFF over that. Also see the `formats.sort()` method.
- `IMAGEIO_REQUEST_TIMEOUT`: Set the timeout of http/ftp request in seconds. If not set, this defaults to 5 seconds.
- `IMAGEIO_USERDIR`: Set the path to the default user directory. If not given, imageio will try ~ and if that's not available `/var/tmp`.

1.8 Upgrading to ImageIO

1.8.1 Transitioning from Scipy's imread

Scipy has depreciated their image I/O functionality.

This document is intended to help people coming from Scipy to adapt to ImageIO's *`imread`* function. We recommend reading the *user api* and checking out some *examples* to get a feel of ImageIO.

ImageIO makes use of a variety of backends to support reading images (and volumes/movies) from many different formats. One of these backends is Pillow, which is the same library as used by Scipy's `imread`. This means that all image formats that were supported by scipy are supported by ImageIO. At the same time, Pillow is not the only backend of ImageIO, and those other backends give you give you access to additional formats. To manage this, ImageIO automatically selects the right backend to use based on the source to read (which you can of course control manually, should you wish to do so).

In short terms: In most places where you used Scipy's imread, ImageIO's imread is a drop-in replacement and ImageIO provides the same functionality as Scipy in these cases. In some cases, you will need to update the keyword arguments used.

- Instead of `mode`, use the `pilmode` keyword argument.
- Instead of `flatten`, use the `as_gray` keyword argument.
- The documentation for the above arguments is not on `imread`, but on the docs of the individual plugins, e.g. *`Pillow`*.
- ImageIO's functions all return numpy arrays, albeit as a subclass (so that meta data can be attached).

1.8.2 Depreciating Python 2.7

Python 2.7 is deprecated from 2020. For more information on the scientific Python ecosystem's transition to Python3 only, see the python3-statement.

Imageio 2.6.x is the last release to support Python 2.7. This release will remain available on Pypi and conda-forge. The *py2* branch may be used to continue supporting 2.7, but one should not expect (free) contributions from the imageio developers.

For more information on porting your code to run on Python 3, see the python3-howto.

CHAPTER

TWO

SUPPORTED FORMATS

Note: If you just want to know if a specific extension/format is supported you can search this page using `Ctrl+F` and then type the name of the extension or format.

ImageIO reads and writes images by deligating your request to one of many backends. Example backends are pillow, ffmpeg, tifffile among others. Each backend supports its own set of formats, which is how ImageIO manages to support so many of them.

To help you navigate this format jungle, ImageIO provides various curated lists of formats depending on your use-case

2.1 Formats by Plugin

Below you can find a list of each plugin that exists in ImageIO together with the formats that this plugin supports. This can be useful, for example, if you have to decide which plugins to install and/or depend on in your project.

FreeImage tif, tiff, jpeg, jpg, bmp, png, bw, dds, gif, ico, j2c, j2k, jp2,pbm, pcd, PCX, pgm, ppm, psd, ras, rgb, rgba, sgi, tga, xbm, xpm, pic, raw, 3fr, arw, bay, bmq, cap, cine, cr2, crw, cs1, cut, dc2, dcr, dng, drf, dsc, erf, exr, fff, g3, hdp, hdr, ia, iff, iiq, jif, jng, jpe, jxr, k25, kc2, kdc, koa, lbm, mdc, mef, mos, mrw, nef, nrw, orf, pct, pef, pfm, pict, ptx, pxn, qtk, raf, rdc, rw2, rwl, rwz, sr2, srf, srw, sti, targa, wap, wbm, wbmp, wdp, webp

Pillow tif, tiff, jpeg, jpg, bmp, png, bw, dds, gif, ico, j2c, j2k, jp2, pbm, pcd, PCX, pgm, ppm, psd, ras, rgb, rgba, sgi, tga, xbm, xpm, fit, fits, bufr, CLIPBOARDGRAB, cur, dcx, DIB, emf, eps, flc, fli, fpx, ftc, ftu, gbr, grib, h5, hdf, icns, iim, im, IMT, jfif, jpc, jpf, jpx, MCIDAS, mic, mpo, msp, ps, pxr, SCREENGRAB, SPIDER, wmf, XVTHUMB

ITK tif, tiff, jpeg, jpg, bmp, png, pic, img, lsm, dcm, dicom, gdcm, gipl, hdf5, hdr, img.gz, ipl, mgh, mha, mhd, mnc, mnc2, nhdr, nia, nii, nii.gz, nrrd, vtk

FFMPEG avi, mkv, mov, mp4, mpeg, mpg, WEBCAM, webm, wmv

GDAL tif, tiff, jpeg, jpg, img, ecw

tifffile tif,tiff,lsm,stk

FITS fit,fits,fts,fz

DICOM dcm,ct,mri

Lytro raw,lfp,lfr

FEI/SEM tif, tiff

Numpy npz

BSDF bsdf

SPE spe

SWF swf

2.2 Video Formats

Below you can find an alphabetically sorted list of the video extensions/formats that ImageIO is aware of. If an extension is listed here, it is supported. If an extension is not listed here, it may still be supported if one of the backends supports the extension/format. If you encounter the latter, please create a new issue so that we can keep below list up to date and add support for any missing formats.

Each entry in the list below follows the following format:

```
<extension> (<format name>): <plugin> <plugin> ...
```

where <plugin> is the name of a plugin that can handle the format. If you wish to use a specific plugin to load a format, you would use the name as specified. For example, if you have a MOV file that you wish to open with FFMPEG and the <plugin> is called FFMPEG you would call:

```
iio.imread("image.mov", format="FFMPEG")
```

Format List

Note: To complete this list we are looking for each format's full name and a link to the spec. If you come across this information, please consider sharing it by creating a new issue.

- **avi** (Audio Video Interleave): *FFMPEG*
- **gif** (Graphics Interchange Format): *GIF-FI GIF-PIL*
- **mkv** (Matroska Multimedia Container): *FFMPEG*
- **mov** (QuickTime File Format): *FFMPEG*
- **mp4** (MPEG-4 Part 14): *FFMPEG*
- **mpeg** (Moving Picture Experts Group): *FFMPEG*
- **mpg** (Moving Picture Experts Group): *FFMPEG*
- **webm**: *FFMPEG*
- **wmv** (Windows Media Video): *FFMPEG*

2.3 All Formats

Below you can find an alphabetically sorted list of *all* extensions/file-formats that ImageIO is aware of. If an extension is listed here, it is supported. If an extension is not listed here, it may still be supported if one of the backends supports the extension/format. If you encoutner the latter, please create a new issue so that we can keep below list up to date and add support for any missing formats.

Each entry in the list below follows the following format:

```
extension (format_name): plugin1 plugin2 ...
```

where the plugins refer to imageio plugins that can handle the format. If you wish to use a specific plugin to load a format, you would use the name as specified here. For example, if you have a PNG file that you wish to open with pillow you would call:

```
iio.imread("image.png", format="PNG-PIL")
```

Format List

Note: To complete this list we are looking for each format's full name and a link to the spec. If you come across this information, please consider sharing it by creating a new issue.

- **3fr** (Hasselblad raw): *RAW-FI*
- **DIB** (Windows Bitmap): *DIB-PIL*
- **IMT** (IM Tools): *IMT-PIL*
- **MCIDAS** (McIdas area file): *MCIDAS-PIL*
- **PCX** (Zsoft Paintbrush): *PCX-FI PCX-PIL*
- **SPIDER**: *SPIDER-PIL*
- **XVTHUMB** (Thumbnail Image): *XVTHUMB-PIL*
- **arw** (Sony alpha): *RAW-FI*
- **avi** (Audio Video Interleave): *FFMPEG*
- **bay** (Casio raw format): *RAW-FI*
- **bmp** (Bitmap): *BMP-PIL BMP-FI itk*
- **bmq** (Re-Volt mipmap): *RAW-FI*
- **bsdf** (Binary Structured Data Format): *BSDF*
- **bufr** (Binary Universal Form for the Representation of meteorological data): *BUFR-PIL*
- **bw** (Silicon Graphics Image): *SGI-PI SGI-FI*
- **cap** (Scirra Construct): *RAW-FI*
- **cine** (AMETEK High Speed Camera Format): *RAW-FI*
- **cr2**: *RAW-FI*
- **crw**: *RAW-FI*
- **cs1**: *RAW-FI*
- **ct** (Computerized Tomography): *DICOM*
- **cur** (Windows Cursor Icons): *CUR-PIL*
- **cut** (Dr. Halo): *CUT-FI*
- **dc2**: *RAW-FI*
- **dcm** (DICOM file format): *itk DICOM*

- **dcr**: *RAW-FI*
- **dcx** (Intel DCX): *DCX-PIL*
- **dds** (DirectX Texture Container): *DDS-FI DDS-PIL*
- **dicom** (DICOM file format): *itk*
- **dng**: *RAW-FI*
- **drf**: *RAW-FI*
- **dsc**: *RAW-FI*
- **ecw** (Enhanced Compression Wavelet): *GDAL*
- **emf** (Windows Metafile): *WMF-PIL*
- **eps** (Encapsulated Postscript): *EPS-PIL*
- **erf**: *RAW-FI*
- **exr** (ILM OpenEXR): *EXR-FI*
- **fff**: *RAW-FI*
- **fit** (Flexible Image Transport System File): *FITS-PIL fits*
- **fits** (Flexible Image Transport System File): *FITS-PIL fits*
- **flc** (Autodesk FLC Animation): *FLI-PIL*
- **fli** (Autodesk FLI Animation): *FLI-PIL*
- **fpx** (Kodak FlashPix): *FPX-PIL*
- **ftc** (Independence War 2: Edge Of Chaos Texture Format): *FTEX-PIL*
- **fts** (Flexible Image Transport System File): *fits*
- **ftu** (Independence War 2: Edge Of Chaos Texture Format): *FTEX-PIL*
- **fz** (Flexible Image Transport System File): *fits*
- **g3** (Raw fax format CCITT G.3): *G3-FI*
- **gbr** (GIMP brush file): *GBR-PIL*
- **gdcm** (Grassroots DICOM): *itk*
- **gif** (Graphics Interchange Format): *GIF-FI GIF-PIL*
- **gipl** (UMDS GIPL): *itk*
- **grib** (gridded meteorological data): *GRIB-PIL*
- **h5** (Hierarchical Data Format 5): *HDF5-PIL*
- **hdf** (Hierarchical Data Format 5): *HDF5-PIL*
- **hdf5** (Hierarchical Data Format 5): *itk*
- **hdp** (JPEG Extended Range): *JPEG-XR-FI*
- **hdr** (High Dynamic Range Image): *HDR-FI itk*
- **ia**: *RAW-FI*
- **icns** (Mac OS Icon File): *ICNS-PIL*
- **ico** (Windows Icon File): *ICO-FI ICO-PIL*

- **iff** (ILBM Interleaved Bitmap): *IFF-FI*
- **iim** (IPTC/NAA): *IPTC-PIL*
- **iiq**: *RAW-FI*
- **im** (IFUNC Image Memory): *IM-PIL*
- **img**: *itk GDAL*
- **img.gz**: *itk*
- **ipl** (Image Processing Lab): *itk*
- **j2c** (JPEG 2000): *J2K-FI JPEG2000-PIL*
- **j2k** (JPEG 2000): *J2K-FI JPEG2000-PIL*
- **jfif** (JPEG): *JPEG-PIL*
- **jif** (JPEG): *JPEG-FI*
- **jng** (JPEG Network Graphics): *JNG-FI*
- **jp2** (JPEG 2000): *JP2-FI JPEG2000-PIL*
- **jpc** (JPEG 2000): *JPEG2000-PIL*
- **jpe** (JPEG): *JPEG-FI JPEG-PIL*
- **jpeg** (Joint Photographic Experts Group): *JPEG-FI JPEG-PIL itk GDAL*
- **jpf** (JPEG 2000): *JPEG2000-PIL*
- **jpg** (Joint Photographic Experts Group): *JPEG-FI JPEG-PIL itk GDAL*
- **jpx** (JPEG 2000): *JPEG2000-PIL*
- **jxr** (JPEG Extended Range): *JPEG-XR-FI*
- **k25**: *RAW-FI*
- **kc2**: *RAW-FI*
- **kdc**: *RAW-FI*
- **koa** (C64 Koala Graphics): *KOALA-FI*
- **lbm** (ILBM Interleaved Bitmap): *IFF-FI*
- **lfp** (Lytro F01): *LYTRO-LFP*
- **lfr** (Lytro Illum): *LYTRO-LFR*
- **lsm** (ZEISS LSM): *itk tiff*
- **mdc**: *RAW-FI*
- **mef**: *RAW-FI*
- **mgh** (FreeSurfer File Format): *itk*
- **mha** (ITK MetaImage): *itk*
- **mhd** (ITK MetaImage Header): *itk*
- **mic** (Microsoft Image Composer): *MIC-PIL*
- **mkv** (Matroska Multimedia Container): *FFMPEG*
- **mnc** (Medical Imaging NetCDF): *itk*

- **mnc2** (Medical Imaging NetCDF 2): *itk*
- **mos** (Leaf Raw Image Format): *RAW-FI*
- **mov** (QuickTime File Format): *FFMPEG*
- **mp4** (MPEG-4 Part 14): *FFMPEG*
- **mpeg** (Moving Picture Experts Group): *FFMPEG*
- **mpg** (Moving Picture Experts Group): *FFMPEG*
- **mpo** (JPEG Multi-Picture Format): *MPO-PIL*
- **mri** (Magnetic resonance imaging): *DICOM*
- **mrw**: *RAW-FI*
- **msp** (Windows Paint): *MSP-PIL*
- **nef**: *RAW-FI*
- **nhdr**: *itk*
- **nia**: *itk*
- **nii**: *itk*
- **nii.gz** (nii.gz): *itk*
- **npz** (Numpy Array): *npz*
- **nrrd**: *itk*
- **nrw**: *RAW-FI*
- **orf**: *RAW-FI*
- **pbm** (Portable Bitmap): *PGM-FI PGMRAW-FI*
- **pbm** (Pbmplus image): *PPM-PIL*
- **pbm** (Pbmplus image): *PPM-PIL PPM-FI*
- **pcd** (Kodak PhotoCD): *PCD-FI PCD-PIL*
- **pct** (Macintosh PICT): *PICT-PIL*
- **pef**: *RAW-FI*
- **pfm**: *PFM-FI*
- **pgm** (Portable Greymap): *PGM-FI PGMRAW-FI*
- **pic** (Macintosh PICT): *PICT-PIL itk*
- **pict** (Macintosh PICT): *PICT-PIL*
- **png** (Portable Network Graphics): *PNG-FI PNG-PIL itk*
- **ppm** (Pbmplus image): *PPM-PIL*
- **ppm** (Portable Pixelmap (ASCII)): *PPM-FI*
- **ppm** (Portable Pixelmap (Raw)): *PPMRAW-FI*
- **ps** (Ghostscript): *EPS-PIL*
- **psd** (Adope Photoshop 2.5 and 3.0): *PSD-PIL PSD-FI*
- **ptx**: *RAW-FI*

- **pxn**: *RAW-FI*
- **pxr** (PIXAR raster image): *PIXAR-PIL*
- **qtk**: *RAW-FI*
- **raf**: *RAW-FI*
- **ras** (Sun Raster File): *SUN-PIL RAS-FI*
- **raw**: *RAW-FI LYTRO-ILLUM-RAW LYTRO-F01-RAW*
- **rdc**: *RAW-FI*
- **rgb** (Silicon Graphics Image): *SGI-PIL*
- **rgba** (Silicon Graphics Image): *SGI-PIL*
- **rw2**: *RAW-FI*
- **rwl**: *RAW-FI*
- **rwz**: *RAW-FI*
- **sgi** (Silicon Graphics Image): *SGI-PIL*
- **spe** (SPE File Format): *SPE*
- **sr2**: *RAW-FI*
- **srf**: *RAW-FI*
- **srw**: *RAW-FI*
- **sti**: *RAW-FI*
- **stk**: *tiff*
- **swf** (Shockwave Flash): *SWF*
- **targa** (Truevision TGA): *TARGA-FI*
- **tga** (Truevision TGA): *TGA-PIL TARGA-FI*
- **tif** (Tagged Image File): *tiff TIFF-PIL TIFF-FI FEI itk GDAL*
- **tiff** (Tagged Image File Format): *tiff TIFF-PIL TIFF-FI FEI itk GDAL*
- **vtk**: *itk*
- **wap** (Wireless Bitmap): *WBMP-FI*
- **wbm** (Wireless Bitmap): *WBMP-FI*
- **wbmp** (Wireless Bitmap): *WBMP-FI*
- **wdp** (JPEG Extended Range): *JPEG-XR-FI*
- **webm**: *FFMPEG*
- **webp** (Google WebP): *WEBP-FI*
- **wmf** (Windows Meta File): *WMF-PIL*
- **wmv** (Windows Media Video): *FFMPEG*
- **xbm** (X11 Bitmap): *XBM-PIL XBM-FI*
- **xpm** (X11 Pixel Map): *XPM-PIL XPM-FI*

CHAPTER

THREE

API REFERENCE

ImageIO's API follows the usual idea of choosing sensible defaults for the average use, but giving you fine grained control

3.1 Core API (Basic Usage)

The API documented in this section is shared by all plugins

3.1.1 Core API v2.9

These functions represent imageio's main interface for the user. They provide a common API to read and write image data for a large variety of formats. All read and write functions accept keyword arguments, which are passed on to the backend that does the actual work. To see what keyword arguments are supported by a specific format, use the *help()* function.

Note: All read-functions return images as numpy arrays, and have a `meta` attribute; the meta-data dictionary can be accessed with `im.meta`. To make this work, imageio actually makes use of a subclass of `np.ndarray`. If needed, the image can be converted to a plain numpy array using `np.asarray(im)`.

`imageio.help`(*name=None*)

Print the documentation of the format specified by name, or a list of supported formats if name is omitted.

Parameters

name [str] Can be the name of a format, a filename extension, or a full filename. See also the *formats page*.

`imageio.show_formats`()

Show a nicely formatted list of available formats

Functions for reading

imageio.imread(uri[, format])	Reads an image from the specified file.
imageio.mimread(uri[, format, memtest])	Reads multiple images from the specified file.
imageio.volread(uri[, format])	Reads a volume from the specified file.
imageio.mvolread(uri[, format, memtest])	Reads multiple volumes from the specified file.

imageio.imread

imageio.**imread**(*uri*, *format=None*, ***kwargs*)

Reads an image from the specified file. Returns a numpy array, which comes with a dict of meta data at its 'meta' attribute.

Note that the image data is returned as-is, and may not always have a dtype of uint8 (and thus may differ from what e.g. PIL returns).

Parameters

uri [{str, pathlib.Path, bytes, file}] The resource to load the image from, e.g. a filename, pathlib.Path, http address or file object, see the docs for more info.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the reader. See *help()* to see what arguments are available for a particular format.

imageio.mimread

imageio.**mimread**(*uri*, *format=None*, *memtest='256MB'*, ***kwargs*)

Reads multiple images from the specified file. Returns a list of numpy arrays, each with a dict of meta data at its 'meta' attribute.

Parameters

uri [{str, pathlib.Path, bytes, file}] The resource to load the images from, e.g. a filename,pathlib.Path, http address or file object, see the docs for more info.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

memtest [{bool, int, float, str}] If truthy, this function will raise an error if the resulting list of images consumes greater than the amount of memory specified. This is to protect the system from using so much memory that it needs to resort to swapping, and thereby stall the computer. E.g. `mimread('hunger_games.avi')`.

If the argument is a number, that will be used as the threshold number of bytes.

If the argument is a string, it will be interpreted as a number of bytes with SI/IEC prefixed units (e.g. '1kB', '250MiB', '80.3YB').

- Units are case sensitive
- k, M etc. represent a 1000-fold change, where Ki, Mi etc. represent 1024-fold
- The "B" is optional, but if present, must be capitalised

If the argument is True, the default will be used, for compatibility reasons.

Default: '256MB'

kwargs [...] Further keyword arguments are passed to the reader. See *help()* to see what arguments are available for a particular format.

imageio.volread

imageio.**volread**(*uri*, *format=None*, ***kwargs*)

Reads a volume from the specified file. Returns a numpy array, which comes with a dict of meta data at its 'meta' attribute.

Parameters

uri [{str, pathlib.Path, bytes, file}] The resource to load the volume from, e.g. a filename, pathlib.Path, http address or file object, see the docs for more info.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the reader. See *help()* to see what arguments are available for a particular format.

imageio.mvolread

imageio.**mvolread**(*uri*, *format=None*, *memtest='1GB'*, ***kwargs*)

Reads multiple volumes from the specified file. Returns a list of numpy arrays, each with a dict of meta data at its 'meta' attribute.

Parameters

uri [{str, pathlib.Path, bytes, file}] The resource to load the volumes from, e.g. a filename, pathlib.Path, http address or file object, see the docs for more info.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

memtest [{bool, int, float, str}] If truthy, this function will raise an error if the resulting list of images consumes greater than the amount of memory specified. This is to protect the system from using so much memory that it needs to resort to swapping, and thereby stall the computer. E.g. `mimread('hunger_games.avi')`.

If the argument is a number, that will be used as the threshold number of bytes.

If the argument is a string, it will be interpreted as a number of bytes with SI/IEC prefixed units (e.g. '1kB', '250MiB', '80.3YB').

- Units are case sensitive
- k, M etc. represent a 1000-fold change, where Ki, Mi etc. represent 1024-fold
- The "B" is optional, but if present, must be capitalised

If the argument is True, the default will be used, for compatibility reasons.

Default: '1GB'

kwargs [...] Further keyword arguments are passed to the reader. See *help()* to see what arguments are available for a particular format.

Functions for writing

imageio.imwrite(uri, im[, format])	Write an image to the specified file.
imageio.mimwrite(uri, ims[, format])	Write multiple images to the specified file.
imageio.volwrite(uri, vol[, format])	Write a volume to the specified file.
imageio.mvolwrite(uri, vols[, format])	Write multiple volumes to the specified file.

imageio.imwrite

imageio.**imwrite**(*uri*, *im*, *format=None*, ***kwargs*)
Write an image to the specified file.

Parameters

uri [{str, pathlib.Path, file}] The resource to write the image to, e.g. a filename, pathlib.Path or file object, see the docs for more info.

im [numpy.ndarray] The image data. Must be NxM, NxMx3 or NxMx4.

format [str] The format to use to write the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the writer. See *help()* to see what arguments are available for a particular format.

imageio.mimwrite

imageio.**mimwrite**(*uri*, *ims*, *format=None*, ***kwargs*)
Write multiple images to the specified file.

Parameters

uri [{str, pathlib.Path, file}] The resource to write the images to, e.g. a filename, pathlib.Path or file object, see the docs for more info.

ims [sequence of numpy arrays] The image data. Each array must be NxM, NxMx3 or NxMx4.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the writer. See *help()* to see what arguments are available for a particular format.

imageio.volwrite

imageio.**volwrite**(*uri*, *vol*, *format=None*, ***kwargs*)
Write a volume to the specified file.

Parameters

uri [{str, pathlib.Path, file}] The resource to write the image to, e.g. a filename, pathlib.Path or file object, see the docs for more info.

vol [numpy.ndarray] The image data. Must be NxMxL (or NxMxLxK if each voxel is a tuple).

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the writer. See *help()* to see what arguments are available for a particular format.

imageio.mvolwrite

imageio.**mvolwrite**(*uri*, *vols*, *format=None*, ***kwargs*)
Write multiple volumes to the specified file.

Parameters

uri [{str, pathlib.Path, file}] The resource to write the volumes to, e.g. a filename, pathlib.Path or file object, see the docs for more info.

ims [sequence of numpy arrays] The image data. Each array must be NxMxL (or NxMxLxK if each voxel is a tuple).

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

kwargs [...] Further keyword arguments are passed to the writer. See *help()* to see what arguments are available for a particular format.

More control

For a larger degree of control, imageio provides functions *get_reader()* and *get_writer()*. They respectively return an *Reader* and an *Writer* object, which can be used to read/write data and meta data in a more controlled manner. This also allows specific scientific formats to be exposed in a way that best suits that file-format.

imageio.**get_reader**(*uri*, *format=None*, *mode='?'*, ***kwargs*)
Returns a *Reader* object which can be used to read data and meta data from the specified file.

Parameters

uri [{str, pathlib.Path, bytes, file}] The resource to load the image from, e.g. a filename, pathlib.Path, http address or file object, see the docs for more info.

format [str] The format to use to read the file. By default imageio selects the appropriate for you based on the filename and its contents.

mode [{'i', 'I', 'v', 'V', '?'}] Used to give the reader a hint on what the user expects (default "?"): "i" for an image, "I" for multiple images, "v" for a volume, "V" for multiple volumes, "?" for don't care.

kwargs [...] Further keyword arguments are passed to the reader. See *help()* to see what arguments are available for a particular format.

imageio.**get_writer**(*uri*, *format=None*, *mode='?'*, ***kwargs*)
Returns a *Writer* object which can be used to write data and meta data to the specified file.

Parameters

uri [{str, pathlib.Path, file}] The resource to write the image to, e.g. a filename, pathlib.Path or file object, see the docs for more info.

format [str] The format to use to write the file. By default imageio selects the appropriate for you based on the filename.

mode [{'i', 'I', 'v', 'V', '?'}] Used to give the writer a hint on what the user expects (default '?'): "i" for an image, "I" for multiple images, "v" for a volume, "V" for multiple volumes, "?" for don't care.

kwargs [...] Further keyword arguments are passed to the writer. See *help()* to see what arguments are available for a particular format.

---

class imageio.core.format.**Reader**(*format*, *request*)

The purpose of a reader object is to read data from an image resource, and should be obtained by calling *get_reader()*.

A reader can be used as an iterator to read multiple images, and (if the format permits) only reads data from the file when new data is requested (i.e. streaming). A reader can also be used as a context manager so that it is automatically closed.

Plugins implement Reader's for different formats. Though rare, plugins may provide additional functionality (beyond what is provided by the base reader class).

Attributes

closed Whether the reader/writer is closed.

format The `Format` object corresponding to the current read/write operation.

request The `Request` object corresponding to the current read/write operation.

close()

Flush and close the reader/writer. This method has no effect if it is already closed.

property closed

Whether the reader/writer is closed.

property format

The `Format` object corresponding to the current read/write operation.

get_data(*index*, ***kwargs*)

Read image data from the file, using the image index. The returned image has a 'meta' attribute with the meta data. Raises IndexError if the index is out of range.

Some formats may support additional keyword arguments. These are listed in the documentation of those formats.

get_length()

Get the number of images in the file. (Note: you can also use `len(reader_object)`.)

The result can be:

- 0 for files that only have meta data
- 1 for singleton images (e.g. in PNG, JPEG, etc.)
- N for image series
- inf for streams (series of unknown length)

get_meta_data(*index=None*)

Read meta data from the file. using the image index. If the index is omitted or None, return the file's (global) meta data.

Note that `get_data` also provides the meta data for the returned image as an atrribute of that image.

The meta data is a dict, which shape depends on the format. E.g. for JPEG, the dict maps group names to subdicts and each group is a dict with name-value pairs. The groups represent the different metadata formats (EXIF, XMP, etc.).

get_next_data(***kwargs*)
Read the next image from the series.

Some formats may support additional keyword arguments. These are listed in the documentation of those formats.

iter_data()
Iterate over all images in the series. (Note: you can also iterate over the reader object.)

property request
The `Request` object corresponding to the current read/write operation.

set_image_index(*index*)
Set the internal pointer such that the next call to get_next_data() returns the image specified by the index

class imageio.core.format.**Writer**(*format*, *request*)
The purpose of a writer object is to write data to an image resource, and should be obtained by calling *get_writer()*.

A writer will (if the format permits) write data to the file as soon as new data is provided (i.e. streaming). A writer can also be used as a context manager so that it is automatically closed.

Plugins implement Writer's for different formats. Though rare, plugins may provide additional functionality (beyond what is provided by the base writer class).

Attributes

closed Whether the reader/writer is closed.

format The `Format` object corresponding to the current read/write operation.

request The `Request` object corresponding to the current read/write operation.

append_data(*im*, *meta={}*)
Append an image (and meta data) to the file. The final meta data that is used consists of the meta data on the given image (if applicable), updated with the given meta data.

close()
Flush and close the reader/writer. This method has no effect if it is already closed.

property closed
Whether the reader/writer is closed.

property format
The `Format` object corresponding to the current read/write operation.

property request
The `Request` object corresponding to the current read/write operation.

set_meta_data(*meta*)
Sets the file's (global) meta data. The meta data is a dict which shape depends on the format. E.g. for JPEG the dict maps group names to subdicts, and each group is a dict with name-value pairs. The groups represents the different metadata formats (EXIF, XMP, etc.).

Note that some meta formats may not be supported for writing, and individual fields may be ignored without warning if they are invalid.

3.2 Plugins & Backend Libraries (Advanced Usage)

Sometimes, you need to do more than just "load an image, done". Sometimes you need to use a very specific feature of a very specific backend. ImageIOs allows you to do so by allowing its plugins to extend the core API. Typically this is done in the form of additional keyword arguments (`kwarg`). Below you can find a list of available plugins and which arguments they support.

imageio.plugins.bsdf	Read/Write BSDF files.
imageio.plugins.dicom	Read DICOM files.
imageio.plugins.feisem	Read TIFF from FEI SEM microscopes.
imageio.plugins.ffmpeg	Read/Write video using FFMPEG
imageio.plugins.fits	Read FITS files.
imageio.plugins.freeimage	Read/Write images using FreeImage.
imageio.plugins.gdal	Read GDAL files.
imageio.plugins.lytro	Read LFR files (Lytro Illum).
imageio.plugins.npz	Read/Write NPZ files.
imageio.plugins.pillow	Read/Write images using Pillow/PIL.
imageio.plugins.pillow_legacy	Read/Write images using pillow/PIL (legacy).
imageio.plugins.simpleitk	Read/Write images using SimpleITK.
imageio.plugins.spe	Read SPE files.
imageio.plugins.swf	Read/Write SWF files.
imageio.plugins.tifffile	Read/Write TIFF files.

3.2.1 imageio.plugins.bsdf

Read/Write BSDF files.

Backend Library: internal

The BSDF format enables reading and writing of image data in the BSDF serialization format. This format allows storage of images, volumes, and series thereof. Data can be of any numeric data type, and can optionally be compressed. Each image/volume can have associated meta data, which can consist of any data type supported by BSDF.

By default, image data is lazily loaded; the actual image data is not read until it is requested. This allows storing multiple images in a single file and still have fast access to individual images. Alternatively, a series of images can be read in streaming mode, reading images as they are read (e.g. from http).

BSDF is a simple generic binary format. It is easy to extend and there are standard extension definitions for 2D and 3D image data. Read more at http://bsdf.io.

Parameters

random_access [bool] Whether individual images in the file can be read in random order. Defaults to True for normal files, and to False when reading from HTTP. If False, the file is read in "streaming mode", allowing reading files as they are read, but without support for "rewinding". Note that setting this to True when reading from HTTP, the whole file is read upon opening it (since lazy loading is not possible over HTTP).

compression [int] Use `0` or "no" for no compression, `1` or "zlib" for Zlib compression (same as zip files and PNG), and `2` or "bz2" for Bz2 compression (more compact but slower). Default 1 (zlib). Note that some BSDF implementations may not support compression (e.g. JavaScript).

3.2.2 imageio.plugins.dicom

Read DICOM files.

Backend Library: internal

A format for reading DICOM images: a common format used to store medical image data, such as X-ray, CT and MRI.

This format borrows some code (and ideas) from the pydicom project. However, only a predefined subset of tags are extracted from the file. This allows for great simplifications allowing us to make a stand-alone reader, and also results in a much faster read time.

By default, only uncompressed and deflated transfer syntaxes are supported. If gdcm or dcmtk is installed, these will be used to automatically convert the data. See https://github.com/malaterre/GDCM/releases for installing GDCM.

This format provides functionality to group images of the same series together, thus extracting volumes (and multiple volumes). Using volread will attempt to yield a volume. If multiple volumes are present, the first one is given. Using mimread will simply yield all images in the given directory (not taking series into account).

Parameters

progress [{True, False, BaseProgressIndicator}] Whether to show progress when reading from multiple files. Default True. By passing an object that inherits from BaseProgressIndicator, the way in which progress is reported can be costumized.

3.2.3 imageio.plugins.feisem

Read TIFF from FEI SEM microscopes.

Backend Library: internal

This format is based on *TIFF*, and supports the same parameters. FEI microscopes append metadata as ASCII text at the end of the file, which this reader correctly extracts.

Parameters

discard_watermark [bool] If True (default), discard the bottom rows of the image, which contain no image data, only a watermark with metadata.

watermark_height [int] The height in pixels of the FEI watermark. The default is 70.

See Also

imageio.plugins.tifffile

3.2.4 imageio.plugins.ffmpeg

Read/Write video using FFMPEG

Backend Library: https://github.com/imageio/imageio-ffmpeg

Note: To use this plugin you have to install its backend:

```
pip install imageio[ffmpeg]
```

The ffmpeg format provides reading and writing for a wide range of movie formats such as .avi, .mpeg, .mp4, etc. as well as the ability to read streams from webcams and USB cameras. It is based on ffmpeg and is inspired by/based moviepy by Zulko.

Parameters for reading

fps [scalar] The number of frames per second to read the data at. Default None (i.e. read at the file's own fps). One can use this for files with a variable fps, or in cases where imageio is unable to correctly detect the fps.

loop [bool] If True, the video will rewind as soon as a frame is requested beyond the last frame. Otherwise, IndexError is raised. Default False. Setting this to True will internally call `count_frames()`, and set the reader's length to that value instead of inf.

size [str | tuple] The frame size (i.e. resolution) to read the images, e.g. (100, 100) or "640x480". For camera streams, this allows setting the capture resolution. For normal video data, ffmpeg will rescale the data.

dtype [str | type] The dtype for the output arrays. Determines the bit-depth that is requested from ffmpeg. Supported dtypes: uint8, uint16. Default: uint8.

pixelformat [str] The pixel format for the camera to use (e.g. "yuyv422" or "gray"). The camera needs to support the format in order for this to take effect. Note that the images produced by this reader are always RGB.

input_params [list] List additional arguments to ffmpeg for input file options. (Can also be provided as `ffmpeg_params` for backwards compatibility) Example ffmpeg arguments to use aggressive error handling: ['-err_detect', 'aggressive']

output_params [list] List additional arguments to ffmpeg for output file options (i.e. the stream being read by imageio).

print_info [bool] Print information about the video file as reported by ffmpeg.

Parameters for writing

fps [scalar] The number of frames per second. Default 10.

codec [str] the video codec to use. Default 'libx264', which represents the widely available mpeg4. Except when saving .wmv files, then the defaults is 'msmpeg4' which is more commonly supported for windows

quality [float | None] Video output quality. Default is 5. Uses variable bit rate. Highest quality is 10, lowest is 0. Set to None to prevent variable bitrate flags to FFMPEG so you can manually specify them using output_params instead. Specifying a fixed bitrate using 'bitrate' disables this parameter.

bitrate [int | None] Set a constant bitrate for the video encoding. Default is None causing 'quality' parameter to be used instead. Better quality videos with smaller file sizes will result from using the 'quality' variable bitrate parameter rather than specifiying a fixed bitrate with this parameter.

pixelformat: str The output video pixel format. Default is 'yuv420p' which most widely supported by video players.

input_params [list] List additional arguments to ffmpeg for input file options (i.e. the stream that imageio provides).

output_params [list] List additional arguments to ffmpeg for output file options. (Can also be provided as `ffmpeg_params` for backwards compatibility) Example ffmpeg arguments to use only intra frames and set aspect ratio: ['-intra', '-aspect', '16:9']

ffmpeg_log_level: str Sets ffmpeg output log level. Default is "warning". Values can be "quiet", "panic", "fatal", "error", "warning", "info" "verbose", or "debug". Also prints the FFMPEG command being used by imageio if "info", "verbose", or "debug".

macro_block_size: int Size constraint for video. Width and height, must be divisible by this number. If not divisible by this number imageio will tell ffmpeg to scale the image up to the next closest size divisible by this number. Most codecs are compatible with a macroblock size of 16 (default), some can go smaller (4, 8). To disable this automatic feature set it to None or 1, however be warned many players can't decode videos that are odd in size and some codecs will produce poor results or fail. See https://en.wikipedia.org/wiki/Macroblock.

Notes

If you are using anaconda and `anaconda/ffmpeg` you will not be able to encode/decode H.264 (likely due to licensing concerns). If you need this format on anaconda install `conda-forge/ffmpeg` instead.

You can use the `IMAGEIO_FFMPEG_EXE` environment variable to force using a specific ffmpeg executable.

To get the number of frames before having read them all, you can use the `reader.count_frames()` method (the reader will then use `imageio_ffmpeg.count_frames_and_secs()` to get the exact number of frames, note that this operation can take a few seconds on large files). Alternatively, the number of frames can be estimated from the fps and duration in the meta data (though these values themselves are not always present/reliable).

3.2.5 imageio.plugins.fits

Read FITS files.

Backend Library: Astropy

Note: To use this plugin you have to install its backend:

```
pip install imageio[fits]
```

Flexible Image Transport System (FITS) is an open standard defining a digital file format useful for storage, transmission and processing of scientific and other images. FITS is the most commonly used digital file format in astronomy.

Parameters

cache [bool] If the file name is a URL, *~astropy.utils.data.download_file* is used to open the file. This specifies whether or not to save the file locally in Astropy's download cache (default: *True*).

uint [bool] Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data.

Note, for backward compatibility, the kwarg **uint16** may be used instead. The kwarg was renamed when support was added for integers of any size.

ignore_missing_end [bool] Do not issue an exception when opening a file that is missing an `END` card in the last header.

checksum [bool or str] If *True*, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file. Updates to a file that already has a checksum will preserve and update the existing checksums unless this argument is given a value of 'remove', in which case the CHECKSUM and DATASUM values are not checked, and are removed when saving changes to the file.

disable_image_compression [bool, optional] If *True*, treats compressed image HDU's like normal binary table HDU's.

do_not_scale_image_data [bool] If *True*, image data is not scaled using BSCALE/BZERO values when read.

ignore_blank [bool] If *True*, the BLANK keyword is ignored if present.

scale_back [bool] If *True*, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original BSCALE/BZERO values. This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data.

3.2.6 imageio.plugins.freeimage

Read/Write images using FreeImage.

Backend Library: FreeImage

Note: To use this plugin you have to install its backend:

```
imageio_download_bin freeimage
```

or you can download the backend using the function:

```
imageio.plugins.freeimage.download()
```

Each Freeimage format has the `flags` keyword argument. See the Freeimage documentation for more information.

Parameters

flags [int] A freeimage-specific option. In most cases we provide explicit parameters for influencing image reading.

3.2.7 imageio.plugins.gdal

Read GDAL files.

Backend: GDAL

Note: To use this plugin you have to install its backend:

```
pip install imageio[gdal]
```

Parameters

none

3.2.8 imageio.plugins.lytro

Read LFR files (Lytro Illum).

Backend: internal

Plugin to read Lytro Illum .lfr and .raw files as produced by the Lytro Illum light field camera. It is actually a collection of plugins, each supporting slightly different keyword arguments

Parameters

meta_only [bool] Whether to only read the metadata.

include_thumbnail [bool] (only for lytro-lfr and lytro-lfp) Whether to include an image thumbnail in the metadata.

3.2.9 imageio.plugins.npz

Read/Write NPZ files.

Backend: Numpy

NPZ is a file format by numpy that provides storage of array data using gzip compression. This imageio plugin supports data of any shape, and also supports multiple images per file. However, the npz format does not provide streaming; all data is read/written at once. Further, there is no support for meta data.

See the BSDF format for a similar (but more fully featured) format.

Parameters

None

Notes

This format is not available on Pypy.

3.2.10 imageio.plugins.pillow

Read/Write images using Pillow/PIL.

Backend Library: Pillow

Plugin that wraps the the Pillow library. Pillow is a friendly fork of PIL (Python Image Library) and supports reading and writing of common formats (jpg, png, gif, tiff, ...). For, the complete list of features and supported formats please refer to pillows official docs (see the Backend Library link).

Parameters

request [Request] A request object representing the resource to be operated on.

Methods

PillowPlugin.read(*[, index, mode, rotate, ...])	Parses the given URI and creates a ndarray from it.
PillowPlugin.write(image, *[, mode, format])	Write an ndimage to the URI specified in path.
PillowPlugin.iter(*[, mode, rotate, apply_gamma])	Iterate over all ndimages/frames in the URI
PillowPlugin.get_meta(*[, index])	Read ndimage metadata from the URI

imageio.plugins.pillow.PillowPlugin.read

PillowPlugin.**read**(*, *index=None*, *mode=None*, *rotate=False*, *apply_gamma=False*)

Parses the given URI and creates a ndarray from it.

Parameters

index [{integer}] If the URI contains a list of ndimages (multiple frames) return the index-th image/frame. If None, read all ndimages (frames) in the URI and attempt to stack them along a new 0-th axis (equivalent to np.stack(imgs, axis=0))

mode [{str, None}] Convert the image to the given mode before returning it. If None, the mode will be left unchanged. Possible modes can be found at: https://pillow.readthedocs.io/en/stable/handbook/concepts.html#modes

rotate [{bool}] If set to `True` and the image contains an EXIF orientation tag, apply the orientation before returning the ndimage.

apply_gamma [{bool}] If `True` and the image contains metadata about gamma, apply gamma correction to the image.

Returns

ndimage [ndarray] A numpy array containing the loaded image data

Notes

If you open a GIF - or any other format using color pallets - you may wish to manually set the *mode* parameter. Otherwise, the numbers in the returned image will refer to the entries in the color pallet, which is discarded during conversion to ndarray.

imageio.plugins.pillow.PillowPlugin.write

PillowPlugin.**write**(*image*, *, *mode='RGB'*, *format=None*, ***kwargs*)

Write an ndimage to the URI specified in path.

If the URI points to a file on the current host and the file does not yet exist it will be created. If the file exists already, it will be appended if possible; otherwise, it will be replaced.

If necessary, the image is broken down along the leading dimension to fit into individual frames of the chosen format. If the format doesn't support multiple frames, and IOError is raised.

Parameters

image [ndarray] The ndimage to write.

mode [{str}] Specify the image's color format; default is RGB. Possible modes can be found at: https://pillow.readthedocs.io/en/stable/handbook/concepts.html#modes

format [{str, None}] Optional format override. If omitted, the format to use is determined from the filename extension. If a file object was used instead of a filename, this parameter must always be used.

kwargs [...] Extra arguments to pass to pillow. If a writer doesn't recognise an option, it is silently ignored. The available options are described in pillow's image format documentation for each writer.

imageio.plugins.pillow.PillowPlugin.iter

`PillowPlugin.iter`(*, *mode=None*, *rotate=False*, *apply_gamma=False*)

Iterate over all ndimages/frames in the URI

Parameters

mode [{str, None}] Convert the image to the given mode before returning it. If None, the mode will be left unchanged. Possible modes can be found at: https://pillow.readthedocs.io/en/stable/handbook/concepts.html#modes

rotate [{bool}] If set to `True` and the image contains an EXIF orientation tag, apply the orientation before returning the ndimage.

apply_gamma [{bool}] If `True` and the image contains metadata about gamma, apply gamma correction to the image.

imageio.plugins.pillow.PillowPlugin.get_meta

`PillowPlugin.get_meta`(*, *index=None*)

Read ndimage metadata from the URI

Parameters

index [{integer, None}] If the URI contains a list of ndimages return the metadata corresponding to the index-th image. If None, return the metadata for the last read ndimage/frame.

3.2.11 imageio.plugins.pillow_legacy

Read/Write images using pillow/PIL (legacy).

Backend Library: Pillow

Pillow is a friendly fork of PIL (Python Image Library) and supports reading and writing of common formats (jpg, png, gif, tiff, ...). While these docs provide an overview of some of its features, pillow is constantly improving. Hence, the complete list of features can be found in pillows official docs (see the Backend Library link).

Parameters for Reading

pilmode [str] (Available for all formates except GIF-PIL) From the Pillow documentation:

- 'L' (8-bit pixels, grayscale)
- 'P' (8-bit pixels, mapped to any other mode using a color palette)
- 'RGB' (3x8-bit pixels, true color)
- 'RGBA' (4x8-bit pixels, true color with transparency mask)
- 'CMYK' (4x8-bit pixels, color separation)
- 'YCbCr' (3x8-bit pixels, color video format)
- 'I' (32-bit signed integer pixels)
- 'F' (32-bit floating point pixels)

PIL also provides limited support for a few special modes, including 'LA' ('L' with alpha), 'RGBX' (true color with padding) and 'RGBa' (true color with premultiplied alpha).

When translating a color image to grayscale (mode 'L', 'I' or 'F'), the library uses the ITU-R 601-2 luma transform:

```
L = R * 299/1000 + G * 587/1000 + B * 114/1000
```

as_gray [bool] (Available for all formates except GIF-PIL) If True, the image is converted using mode 'F'. When *mode* is not None and *as_gray* is True, the image is first converted according to *mode*, and the result is then "flattened" using mode 'F'.

ignoregamma [bool] (Only available in PNG-PIL) Avoid gamma correction. Default True.

exifrotate [bool] (Only available in JPEG-PIL) Automatically rotate the image according to exif flag. Default True.

Parameters for saving

optimize [bool] (Only available in PNG-PIL) If present and true, instructs the PNG writer to make the output file as small as possible. This includes extra processing in order to find optimal encoder settings.

transparency: (Only available in PNG-PIL) This option controls what color image to mark as transparent.

dpi: tuple of two scalars (Only available in PNG-PIL) The desired dpi in each direction.

pnginfo: PIL.PngImagePlugin.PngInfo (Only available in PNG-PIL) Object containing text tags.

compress_level: int (Only available in PNG-PIL) ZLIB compression level, a number between 0 and 9: 1 gives best speed, 9 gives best compression, 0 gives no compression at all. Default is 9. When `optimize` option is True `compress_level` has no effect (it is set to 9 regardless of a value passed).

compression: int (Only available in PNG-PIL) Compatibility with the freeimage PNG format. If given, it overrides compress_level.

icc_profile: (Only available in PNG-PIL) The ICC Profile to include in the saved file.

bits (experimental): int (Only available in PNG-PIL) This option controls how many bits to store. If omitted, the PNG writer uses 8 bits (256 colors).

quantize: (Only available in PNG-PIL) Compatibility with the freeimage PNG format. If given, it overrides bits. In this case, given as a number between 1-256.

dictionary (experimental): dict (Only available in PNG-PIL) Set the ZLIB encoder dictionary.

prefer_uint8: bool (Only available in PNG-PIL) Let the PNG writer truncate uint16 image arrays to uint8 if their values fall within the range [0, 255]. Defaults to true for legacy compatibility, however it is recommended to set this to false to avoid unexpected behavior when saving e.g. weakly saturated images.

quality [scalar] (Only available in JPEG-PIL) The compression factor of the saved image (1..100), higher numbers result in higher quality but larger file size. Default 75.

progressive [bool] (Only available in JPEG-PIL) Save as a progressive JPEG file (e.g. for images on the web). Default False.

optimize [bool] (Only available in JPEG-PIL) On saving, compute optimal Huffman coding tables (can reduce a few percent of file size). Default False.

dpi [tuple of int] (Only available in JPEG-PIL) The pixel density, `(x,y)`.

icc_profile [object] (Only available in JPEG-PIL) If present and true, the image is stored with the provided ICC profile. If this parameter is not provided, the image will be saved with no profile attached.

exif [dict] (Only available in JPEG-PIL) If present, the image will be stored with the provided raw EXIF data.

subsampling [str] (Only available in JPEG-PIL) Sets the subsampling for the encoder. See Pillow docs for details.

qtables [object] (Only available in JPEG-PIL) Set the qtables for the encoder. See Pillow docs for details.

quality_mode [str] (Only available in JPEG2000-PIL) Either *"rates"* or *"dB"* depending on the units you want to use to specify image quality.

quality [float] (Only available in JPEG2000-PIL) Approximate size reduction (if quality mode is *rates*) or a signal to noise ratio in decibels (if quality mode is *dB*).

loop [int] (Only available in GIF-PIL) The number of iterations. Default 0 (meaning loop indefinitely).

duration [{float, list}] (Only available in GIF-PIL) The duration (in seconds) of each frame. Either specify one value that is used for all frames, or one value for each frame. Note that in the GIF format the duration/delay is expressed in hundredths of a second, which limits the precision of the duration.

fps [float] (Only available in GIF-PIL) The number of frames per second. If duration is not given, the duration for each frame is set to 1/fps. Default 10.

palettesize [int] (Only available in GIF-PIL) The number of colors to quantize the image to. Is rounded to the nearest power of two. Default 256.

subrectangles [bool] (Only available in GIF-PIL) If True, will try and optimize the GIF by storing only the rectangular parts of each frame that change with respect to the previous. Default False.

Notes

To enable JPEG 2000 support, you need to build and install the OpenJPEG library, version 2.0.0 or higher, before building the Python Imaging Library. Windows users can install the OpenJPEG binaries available on the OpenJPEG website, but must add them to their PATH in order to use PIL (if you fail to do this, you will get errors about not being able to load the `_imaging` DLL).

GIF images read with this plugin are always RGBA. The alpha channel is ignored when saving RGB images.

3.2.12 imageio.plugins.simpleitk

Read/Write images using SimpleITK.

Backend: Insight Toolkit

Note: To use this plugin you have to install its backend:

```
pip install imageio[itk]
```

The ItkFormat uses the ITK or SimpleITK library to support a range of ITK-related formats. It also supports a few common formats (e.g. PNG and JPEG).

Parameters

None

3.2.13 imageio.plugins.spe

Read SPE files.

Backend: internal

This plugin supports reading files saved in the Princeton Instruments SPE file format.

Parameters for reading

char_encoding [str] Character encoding used to decode strings in the metadata. Defaults to "latin1".

check_filesize [bool] The number of frames in the file is stored in the file header. However, this number may be wrong for certain software. If this is *True* (default), derive the number of frames also from the file size and raise a warning if the two values do not match.

sdt_meta [bool] If set to *True* (default), check for special metadata written by the *SDT-control* software. Does not have an effect for files written by other software.

Metadata for reading

ROIs [list of dict] Regions of interest used for recording images. Each dict has the "top_left" key containing x and y coordinates of the top left corner, the "bottom_right" key with x and y coordinates of the bottom right corner, and the "bin" key with number of binned pixels in x and y directions.

comments [list of str] The SPE format allows for 5 comment strings of 80 characters each.

controller_version [int] Hardware version

logic_output [int] Definition of output BNC

amp_hi_cap_low_noise [int] Amp switching mode

mode [int] Timing mode

exp_sec [float] Alternative exposure in seconds

date [str] Date string

detector_temp [float] Detector temperature

detector_type [int] CCD / diode array type

st_diode [int] Trigger diode

delay_time [float] Used with async mode

shutter_control [int] Normal, disabled open, or disabled closed

absorb_live [bool] on / off

absorb_mode [int] Reference strip or file

can_do_virtual_chip [bool] True or False whether chip can do virtual chip

threshold_min_live [bool] on / off

threshold_min_val [float] Threshold minimum value

threshold_max_live [bool] on / off

threshold_max_val [float] Threshold maximum value

time_local [str] Experiment local time

time_utc [str] Experiment UTC time

adc_offset [int] ADC offset

adc_rate [int] ADC rate

adc_type [int] ADC type

adc_resolution [int] ADC resolution

adc_bit_adjust [int] ADC bit adjust

gain [int] gain

sw_version [str] Version of software which created this file

spare_4 [bytes] Reserved space

readout_time [float] Experiment readout time

type [str] Controller type

clockspeed_us [float] Vertical clock speed in microseconds

readout_mode [[“full frame”, “frame transfer”, “kinetics”, “”]] Readout mode. Empty string means that this was not set by the Software.

window_size [int] Window size for Kinetics mode

file_header_ver [float] File header version

chip_size [[int, int]] x and y dimensions of the camera chip

virt_chip_size [[int, int]] Virtual chip x and y dimensions

pre_pixels [[int, int]] Pre pixels in x and y dimensions

post_pixels [[int, int],] Post pixels in x and y dimensions

geometric [list of {“rotate”, “reverse”, “flip”}] Geometric operations

sdt_major_version [int] (only for files created by SDT-control) Major version of SDT-control software

sdt_minor_version [int] (only for files created by SDT-control) Minor version of SDT-control software

sdt_controller_name [str] (only for files created by SDT-control) Controller name

exposure_time [float] (only for files created by SDT-control) Exposure time in seconds

color_code [str] (only for files created by SDT-control) Color channels used

detection_channels [int] (only for files created by SDT-control) Number of channels

background_subtraction [bool] (only for files created by SDT-control) Whether background subtraction war turned on

em_active [bool] (only for files created by SDT-control) Whether EM was turned on

em_gain [int] (only for files created by SDT-control) EM gain

modulation_active [bool] (only for files created by SDT-control) Whether laser modulation ("attenuate") was turned on

pixel_size [float] (only for files created by SDT-control) Camera pixel size

sequence_type [str] (only for files created by SDT-control) Type of sequnce (standard, TOCCSL, arbitrary, ...)

grid [float] (only for files created by SDT-control) Sequence time unit ("grid size") in seconds

n_macro [int] (only for files created by SDT-control) Number of macro loops

delay_macro [float] (only for files created by SDT-control) Time between macro loops in seconds

n_mini [int] (only for files created by SDT-control) Number of mini loops

delay_mini [float] (only for files created by SDT-control) Time between mini loops in seconds

n_micro [int (only for files created by SDT-control)] Number of micro loops

delay_micro [float (only for files created by SDT-control)] Time between micro loops in seconds

n_subpics [int] (only for files created by SDT-control) Number of sub-pictures

delay_shutter [float] (only for files created by SDT-control) Camera shutter delay in seconds

delay_prebleach [float] (only for files created by SDT-control) Pre-bleach delay in seconds

bleach_time [float] (only for files created by SDT-control) Bleaching time in seconds

recovery_time [float] (only for files created by SDT-control) Recovery time in seconds

comment [str] (only for files created by SDT-control) User-entered comment. This replaces the "comments" field.

datetime [datetime.datetime] (only for files created by SDT-control) Combines the "date" and "time_local" keys. The latter two plus "time_utc" are removed.

modulation_script [str] (only for files created by SDT-control) Laser modulation script. Replaces the "spare_4" key.

3.2.14 imageio.plugins.swf

Read/Write SWF files.

Backend: internal

Shockwave flash (SWF) is a media format designed for rich and interactive animations. This plugin makes use of this format to store a series of images in a lossless format with good compression (zlib). The resulting images can be shown as an animation using a flash player (such as the browser).

SWF stores images in RGBA format. RGB or grayscale images are automatically converted. SWF does not support meta data.

Parameters for reading

loop [bool] If True, the video will rewind as soon as a frame is requested beyond the last frame. Otherwise, IndexError is raised. Default False.

Parameters for saving

fps [int] The speed to play the animation. Default 12.

loop [bool] If True, add a tag to the end of the file to play again from the first frame. Most flash players will then play the movie in a loop. Note that the imageio SWF Reader does not check this tag. Default True.

html [bool] If the output is a file on the file system, write an html file (in HTML5) that shows the animation. Default False.

compress [bool] Whether to compress the swf file. Default False. You probably don't want to use this. This does not decrease the file size since the images are already compressed. It will result in slower read and write time. The only purpose of this feature is to create compressed SWF files, so that we can test the functionality to read them.

3.2.15 imageio.plugins.tifffile

Read/Write TIFF files.

Backend: internal

Provides support for a wide range of Tiff images using the tifffile backend.

Parameters for reading

offset [int] Optional start position of embedded file. By default this is the current file position.

size [int] Optional size of embedded file. By default this is the number of bytes from the 'offset' to the end of the file.

multifile [bool] If True (default), series may include pages from multiple files. Currently applies to OME-TIFF only.

multifile_close [bool] If True (default), keep the handles of other files in multifile series closed. This is inefficient when few files refer to many pages. If False, the C runtime may run out of resources.

Parameters for saving

bigtiff [bool] If True, the BigTIFF format is used.

byteorder [{'<', '>'}] The endianness of the data in the file. By default this is the system's native byte order.

software [str] Name of the software used to create the image. Saved with the first page only.

Metadata for reading

planar_configuration [{'contig', 'planar'}] Specifies if samples are stored contiguous or in separate planes. By default this setting is inferred from the data shape. 'contig': last dimension contains samples. 'planar': third last dimension contains samples.

resolution_unit [(float, float) or ((int, int), (int, int))] X and Y resolution in dots per inch as float or rational numbers.

compression [int] Value indicating the compression algorithm used, e.g. 5 is LZW, 7 is JPEG, 8 is deflate. If 1, data are uncompressed.

predictor [int] Value 2 indicates horizontal differencing was used before compression, while 3 indicates floating point horizontal differencing. If 1, no prediction scheme was used before compression.

orientation [{'top_left', 'bottom_right', ...}] Oriented of image array.

is_rgb [bool] True if page contains a RGB image.

is_contig [bool] True if page contains a contiguous image.

is_tiled [bool] True if page contains tiled image.

is_palette [bool] True if page contains a palette-colored image and not OME or STK.

is_reduced [bool] True if page is a reduced image of another image.

is_shaped [bool] True if page contains shape in image_description tag.

is_fluoview [bool] True if page contains FluoView MM_STAMP tag.

is_nih [bool] True if page contains NIH image header.

is_micromanager [bool] True if page contains Micro-Manager metadata.

is_ome [bool] True if page contains OME-XML in image_description tag.

is_sgi [bool] True if page contains SGI image and tile depth tags.

is_stk [bool] True if page contains UIC2Tag tag.

is_mdgel [bool] True if page contains md_file_tag tag.

is_mediacy [bool] True if page contains Media Cybernetics Id tag.

is_stk [bool] True if page contains UIC2Tag tag.

is_lsm [bool] True if page contains LSM CZ_LSM_INFO tag.

description [str] Image description

description1 [str] Additional description

is_imagej [None or str] ImageJ metadata

software [str] Software used to create the TIFF file

datetime [datetime.datetime] Creation date and time

Metadata for writing

photometric [{'minisblack', 'miniswhite', 'rgb'}] The color space of the image data. By default this setting is inferred from the data shape.

planarconfig [{'contig', 'planar'}] Specifies if samples are stored contiguous or in separate planes. By default this setting is inferred from the data shape. 'contig': last dimension contains samples. 'planar': third last dimension contains samples.

resolution [(float, float) or ((int, int), (int, int))] X and Y resolution in dots per inch as float or rational numbers.

description [str] The subject of the image. Saved with the first page only.

compress [int] Values from 0 to 9 controlling the level of zlib (deflate) compression. If 0, data are written uncompressed (default).

predictor [bool] If True, horizontal differencing is applied before compression. Note that using an int literal 1 actually means no prediction scheme will be used.

volume [bool] If True, volume data are stored in one tile (if applicable) using the SGI image_depth and tile_depth tags. Image width and depth must be multiple of 16. Few software can read this format, e.g. MeVisLab.

writeshape [bool] If True, write the data shape to the image_description tag if necessary and no other description is given.

extratags: sequence of tuples Additional tags as [(code, dtype, count, value, writeonce)].

code [int] The TIFF tag Id.

dtype [str] Data type of items in 'value' in Python struct format. One of B, s, H, I, 2I, b, h, i, f, d, Q, or q.

count [int] Number of data values. Not used for string values.

value [sequence] 'Count' values compatible with 'dtype'.

writeonce [bool] If True, the tag is written to the first page only.

Notes

Global metadata is stored with the first frame in a TIFF file. Thus calling `Format.Writer.set_meta_data()` after the first frame was written has no effect. Also, global metadata is ignored if metadata is provided via the *meta* argument of `Format.Writer.append_data()`.

If you have installed tifffile as a Python package, imageio will attempt to use that as backend instead of the bundled backend. Doing so can provide access to new performance improvements and bug fixes.

CHAPTER

FOUR

DEVELOPER DOCUMENTATION

4.1 Imageio's developer API

This page lists the developer documentation for imageio. Normal users will generally not need this, except perhaps the `Format` class. All these functions and classes are available in the `imageio.core` namespace.

This subpackage provides the core functionality of imageio (everything but the plugins).

4.2 Creating ImageIO Plugins

Imagio is plugin-based. Every supported format is provided with a plugin. You can write your own plugins to make imageio support additional formats. And we would be interested in adding such code to the imageio codebase!

4.2.1 What is a plugin

In imageio, a plugin provides one or more `Format` objects, and corresponding *`Reader`* and *`Writer`* classes. Each Format object represents an implementation to read/write a particular file format. Its Reader and Writer classes do the actual reading/saving.

The reader and writer objects have a `request` attribute that can be used to obtain information about the read or write `Request`, such as user-provided keyword arguments, as well get access to the raw image data.

4.2.2 Registering

Strictly speaking a format can be used stand alone. However, to allow imageio to automatically select it for a specific file, the format must be registered using `imageio.formats.add_format()`.

Note that a plugin is not required to be part of the imageio package; as long as a format is registered, imageio can use it. This makes imageio very easy to extend.

4.2.3 What methods to implement

Imageio is designed such that plugins only need to implement a few private methods. The public API is implemented by the base classes. In effect, the public methods can be given a descent docstring which does not have to be repeated at the plugins.

For the Format class, the following needs to be implemented/specified:

- The format needs a short name, a description, and a list of file extensions that are common for the file-format in question. These ase set when instantiation the Format object.
- Use a docstring to provide more detailed information about the format/plugin, such as parameters for reading and saving that the user can supply via keyword arguments.
- Implement `_can_read(request)`, return a bool. See also the `Request` class.
- Implement `_can_write(request)`, dito.

For the Format.Reader class:

- Implement `_open(**kwargs)` to initialize the reader. Deal with the user-provided keyword arguments here.
- Implement `_close()` to clean up.
- Implement `_get_length()` to provide a suitable length based on what the user expects. Can be `inf` for streaming data.
- Implement `_get_data(index)` to return an array and a meta-data dict.
- Implement `_get_meta_data(index)` to return a meta-data dict. If index is None, it should return the 'global' meta-data.

For the Format.Writer class:

- Implement `_open(**kwargs)` to initialize the writer. Deal with the user-provided keyword arguments here.
- Implement `_close()` to clean up.
- Implement `_append_data(im, meta)` to add data (and meta-data).
- Implement `_set_meta_data(meta)` to set the global meta-data.

4.2.4 Example / template plugin

```
# -*- coding: utf-8 -*-
# imageio is distributed under the terms of the (new) BSD License.

""" Example plugin. You can use this as a template for your own plugin.
"""

import numpy as np

from .. import formats
from ..core import Format


class DummyFormat(Format):
    """The dummy format is an example format that does nothing.
    It will never indicate that it can read or write a file. When
    explicitly asked to read, it will simply read the bytes. When
```

(continues on next page)

(continued from previous page)

```
    explicitly asked to write, it will raise an error.

    This documentation is shown when the user does ``help('thisformat')``.

    Parameters for reading
    ----------------------
    Specify arguments in numpy doc style here.

    Parameters for saving
    ---------------------
    Specify arguments in numpy doc style here.

    """

    def _can_read(self, request):
        # This method is called when the format manager is searching
        # for a format to read a certain image. Return True if this format
        # can do it.
        #
        # The format manager is aware of the extensions and the modes
        # that each format can handle. It will first ask all formats
        # that *seem* to be able to read it whether they can. If none
        # can, it will ask the remaining formats if they can: the
        # extension might be missing, and this allows formats to provide
        # functionality for certain extensions, while giving preference
        # to other plugins.
        #
        # If a format says it can, it should live up to it. The format
        # would ideally check the request.firstbytes and look for a
        # header of some kind.
        #
        # The request object has:
        # request.filename: a representation of the source (only for reporting)
        # request.firstbytes: the first 256 bytes of the file.
        # request.mode[0]: read or write mode
        # request.mode[1]: what kind of data the user expects: one of 'iIvV?'

        if request.mode[1] in (self.modes + "?"):
            if request.extension in self.extensions:
                return True

    def _can_write(self, request):
        # This method is called when the format manager is searching
        # for a format to write a certain image. It will first ask all
        # formats that *seem* to be able to write it whether they can.
        # If none can, it will ask the remaining formats if they can.
        #
        # Return True if the format can do it.

        # In most cases, this code does suffice:
        if request.mode[1] in (self.modes + "?"):
            if request.extension in self.extensions:
```

(continues on next page)

(continued from previous page)

```
            return True

    # -- reader

    class Reader(Format.Reader):
        def _open(self, some_option=False, length=1):
            # Specify kwargs here. Optionally, the user-specified kwargs
            # can also be accessed via the request.kwargs object.
            #
            # The request object provides two ways to get access to the
            # data. Use just one:
            #  - Use request.get_file() for a file object (preferred)
            #  - Use request.get_local_filename() for a file on the system
            self._fp = self.request.get_file()
            self._length = length  # passed as an arg in this case for testing
            self._data = None

        def _close(self):
            # Close the reader.
            # Note that the request object will close self._fp
            pass

        def _get_length(self):
            # Return the number of images. Can be np.inf
            return self._length

        def _get_data(self, index):
            # Return the data and meta data for the given index
            if index >= self._length:
                raise IndexError("Image index %i > %i" % (index, self._length))
            # Read all bytes
            if self._data is None:
                self._data = self._fp.read()
            # Put in a numpy array
            im = np.frombuffer(self._data, "uint8")
            im.shape = len(im), 1
            # Return array and dummy meta data
            return im, {}

        def _get_meta_data(self, index):
            # Get the meta data for the given index. If index is None, it
            # should return the global meta data.
            return {}  # This format does not support meta data

    # -- writer

    class Writer(Format.Writer):
        def _open(self, flags=0):
            # Specify kwargs here. Optionally, the user-specified kwargs
            # can also be accessed via the request.kwargs object.
            #
            # The request object provides two ways to write the data.
```

(continues on next page)

(continued from previous page)

```
            # Use just one:
            #  - Use request.get_file() for a file object (preferred)
            #  - Use request.get_local_filename() for a file on the system
            self._fp = self.request.get_file()

        def _close(self):
            # Close the reader.
            # Note that the request object will close self._fp
            pass

        def _append_data(self, im, meta):
            # Process the given data and meta data.
            raise RuntimeError("The dummy format cannot write image data.")

        def set_meta_data(self, meta):
            # Process the given meta data (global for all images)
            # It is not mandatory to support this.
            raise RuntimeError("The dummy format cannot write meta data.")


# Register. You register an *instance* of a Format class. Here specify:
format = DummyFormat(
    "dummy",  # short name
    "An example format that does nothing.",  # one line descr.
    ".foobar .nonexistentext",  # list of extensions
    "iI",  # modes, characters in iIvV
)
formats.add_format(format)
```

4.3 Developer Installation

For developers, we provide a simple mechanism to allow importing imageio from the cloned repository <https://github.com/imageio/imageio.git>:

```
git clone https://github.com/imageio/imageio.git
cd imageio
pip install -e .[dev]
```

PYTHON MODULE INDEX

i

INDEX

R

S

V

W