jicbioimage Documentation

Release 0.15.0

Tjelvar Olsson and Matthew Hartley

November 01, 2016

Contents

1 The `jicbioimage` Python package **1**
- 1.1 Features . . . 1
- 1.2 Related packages . . . 1

2 Installation notes **3**
- 2.1 Introduction . . . 3
- 2.2 Manual install . . . 3
- 2.3 Using Docker . . . 5

3 Quick start guide **7**

4 Working with images **11**
- 4.1 Introduction . . . 11
- 4.2 Creating images . . . 11
- 4.3 Accessing png representations of an image . . . 12
- 4.4 Working with stacks of images . . . 12

5 Working with microscopy data **15**
- 5.1 Introduction . . . 15
- 5.2 Image collection classes . . . 15
- 5.3 Obtaining image collections . . . 16
- 5.4 Accessing data from microscopy collections . . . 16

6 Working with transformations **19**
- 6.1 Introduction . . . 19
- 6.2 Pre-built transformations . . . 19
- 6.3 Creating a custom transform . . . 19
- 6.4 Specifying `dtype` contracts . . . 20
- 6.5 Customising the behaviour of the visual audit trail . . . 20

7 IPython integration **21**

8 API documentation **23**

CHAPTER 1

The `jicbioimage` Python package

The `jicbioimage` Python package makes it easy to explore microscopy data in a programmatic fashion. Exploring images via coding means that the exploratory work becomes recorded and reproducible. Furthermore, it makes it easier to convert the exploratory work into (semi) automated analysis work flows.

- Documentation: http://jicbioimage.readthedocs.io
- GitHub: https://github.com/JIC-CSB/jicbioimage
- PyPI: https://pypi.python.org/pypi/jicbioimage
- Free software: MIT License

1.1 Features

- Built in functionality for working with microscopy data
- Automatic generation of audit trails
- IPython integration
- Cross-platform: Linux, Mac and Windows are all supported
- Works with Python 2.7, 3.3 and 3.4

1.2 Related packages

1.2.1 `jicbioimage.core`

- Documentation: http://jicbioimagecore.readthedocs.io
- GitHub: https://github.com/JIC-CSB/jicbioimage.core

1.2.2 `jicbioimage.transform`

- Documentation: http://jicbioimagetransform.readthedocs.io
- GitHub: https://github.com/JIC-CSB/jicbioimage.transform

1.2.3 `jicbioimage.segment`

- Documentation: http://jicbioimagesegment.readthedocs.io
- GitHub: https://github.com/JIC-CSB/jicbioimage.segment

1.2.4 `jicbioimage.illustrate`

- Documentation: http://jicbioimageillustrate.readthedocs.io
- GitHub: https://github.com/JIC-CSB/jicbioimage.illustrate

CHAPTER 2

Installation notes

2.1 Introduction

We have tried to make it as easy as possible to install and use the `jicbioimage` package. Here we detail two options:

- *Manual install*
- *Using Docker*

If you are not familiar with Docker it is probably easiest to start with a manual install. However, if you are already familiar with Docker it is certainly a very convenient way of creating an environment in which to install and run `jicbioimage`.

2.2 Manual install

2.2.1 Install the freeimage library

The `jicbioimage` package depends on freeimage to open image file.

On Linux bases system freeimage can usually be installed using the package manager. For example on Fedora it can be installed using the command below.

```
yum install freeimage
```

On Macs it can be installed using homebrew using the command below.

```
brew install freeimage
```

On Windows download and unzip the FreeImage DLL (in the example below this was done to the root of the `C:` drive). You will then need to add the relevant directory to your `PATH`, for example on a 64-bit system:

```
set PATH=C:\FreeImage\Dist\x64;%PATH%
```

2.2.2 Install the Python package dependencies

The `jicbioimage` package depends on a number of other scientific Python packages. These can be installed using pip.

```
pip install numpy
pip install scipy
pip install scikit-image
```

Although the `jicbioimage` package does not depend on it you may also want to install the IPython notebook. The `jicbioimage` package has been designed to work well with IPython notebook, for example by providing the ability to view `jicbioimage.core.image.Image` and `jicbioimage.core.image.ImageCollection` objects as images and tables of images in the IPython notebook.

```
pip install jupyter
```

2.2.3 Install the BioFormats command line tools

The `jicbioimage` package does not explicitly depend on the BioFormats command line tools. However, they are needed if you want to be able to work with microscopy files.

Download the bftools.zip file from the openmicroscopy website.

Warning: `jicbioimage.core` version 0.14 and greater require BioFormats to be version 5.2.1 or greater to make use of the `-nolookup` option.

You will then need to unzip the file and add it to your `PATH`.

On Linux and Mac based systems unzip the `bftools.zip` file into a memorable location, for example a directory named `tools`.

```
mkdir ~/tools
mv ~/Downloads/bftools.zip ~/tools/
cd ~/tools
unzip bftools.zip
```

Finally add the `bftools` directory to your `PATH`.

```
export PATH=$PATH:~/tools/bftools
```

Note: You may want to add the line above to your `.bashrc` file.

On Windows unzip the `bftools.zip` file to a memorable location, for example the `C:\` drive and set the `PATH` appropriately:

```
set PATH=C:\bftools;%PATH%
```

2.2.4 Install the `jicbioimage` package

Finally install the `jicbioimage` package using `pip`.

```
pip install jicbioimage.core
pip install jicbioimage.transform
pip install jicbioimage.segment
pip install jicbioimage.illustrate
```

2.3 Using Docker

Docker is a technology that allows one to package software along with all its dependencies in a fashion that ensures that the software will always run the same.

For this purpose we have created the `jicscicomp/bioformats` docker image. It contains all the `jicbioimage` dependencies, but not the `jicbioimage` package itself. You can find out how this Docker image was built in the JIC-CSB/scicomp_docker GitHub repository.

If you are already familiar with Docker you can try it out using the command below.

```
$ docker run -it --rm jicscicomp/bioformats
[root@03fda753e799 /]# pip install jicbioimage.core
[root@03fda753e799 /]# pip install jicbioimage.transform
[root@03fda753e799 /]# pip install jicbioimage.segment
[root@03fda753e799 /]# pip install jicbioimage.illustrate
```

If you have not used Docker before you will need to install it. On Mac and Windows download and install the Docker Toolbox. Docker runs natively on Linux, but you will need to install it, see the Docker Installation Notes.

For our image analysis projects we tend to create three directories in our project: `scripts` (where we put the Python scripts), `data` (where we put the raw images) and `output` (where our scripts write their output). When then use a bash script along the lines of the below to launch a container that has access to these directories (read only for the `data` and `scripts` directories).

```
#!/bin/bash

docker run -it --rm -v `pwd`/data:/data:ro  \
                    -v `pwd`/scripts:/scripts:ro  \
                    -v `pwd`/output:/output jicscicomp/bioformats
```

You will have noticed that we did not include the `jicbioimage` package in the `jicscicomp/bioformats` Docker image. The reason for this is that we like to create a specific Docker image for each bioimage analysis project.

If you want to do this you need to create a directory for your Docker image, for example `cell_analysis`. In that directory you then create a `requirements.txt` file with all your Python requirements, e.g.:

```
jicbioimage.core
jicbioimage.transform
jicbioimage.segment
jicbioimage.illustrate
```

And a `Dockerfile` containing the instructions below.

```
FROM jicscicomp/bioformats

COPY requirements.txt .
RUN pip install -r requirements.txt
```

You can now use this setup to build your own Docker image using the command below.

```
docker build -t cell_analysis .
```

Now you can update your bash script to make use of your custom built image, tagged `cell_analysis`.

```
#!/bin/bash

docker run -it --rm -v `pwd`/data:/data:ro  \
                    -v `pwd`/scripts:/scripts:ro  \
                    -v `pwd`/output:/output cell_analysis
```

In our day to day work, providing bioimage analysis support across the John Innes Centre, we have templated much of our initial project setup using Cookiecutter. For some inspiration you may want to install Cookiecutter and create a project setup using our JIC-Image-Analysis/cookiecutter-image-analysis template hosted on GitHub. The command below uses Cookiecutter to create a new project using this template.

```
$ cookiecutter gh:JIC-Image-Analysis/cookiecutter-image-analysis
```

Enjoy!

CHAPTER 3

Quick start guide

Here we illustrate the use of `jicbioimage` to segment an image of Greek coins from Pompeii.

First we load the image from scikit-image as a numpy array.

```
>>> import skimage.data
>>> array = skimage.data.coins()
```

We then create a `jicbioimage.core.image.Image` instance from the array.

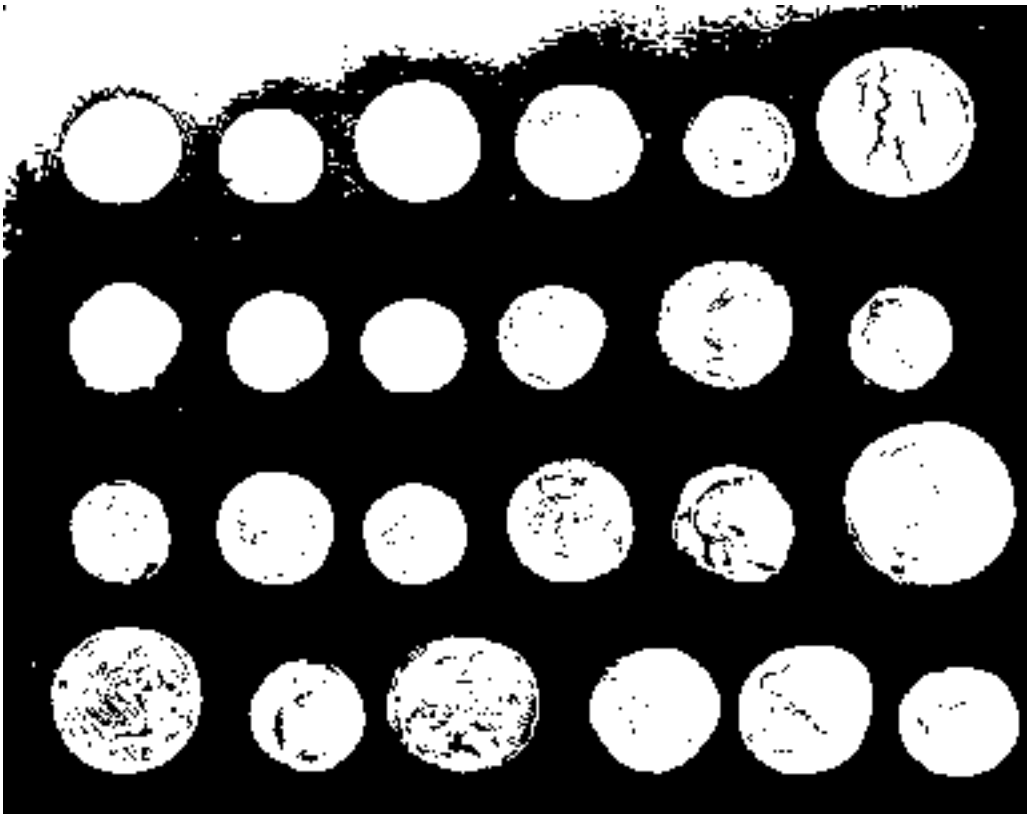
```
>>> from jicbioimage.core.image import Image
>>> image = Image.from_array(array)
```

If using IPython qtconsole/notebook the image can be viewed directly in the interpreter.

```
>>> image
```

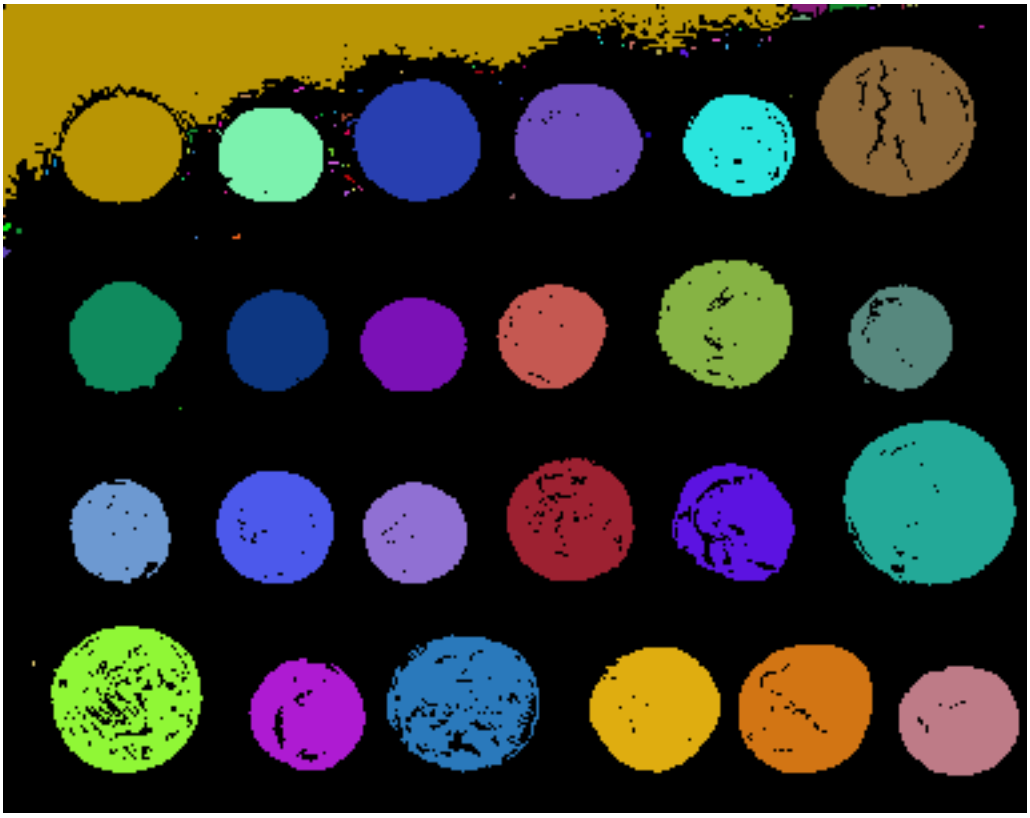
We can now threshold the image, for example using Otsu's method.

```
>>> from jicbioimage.transform import threshold_otsu
>>> image = threshold_otsu(image)
>>> image
```



Let us segment the thresholded image into connected components.

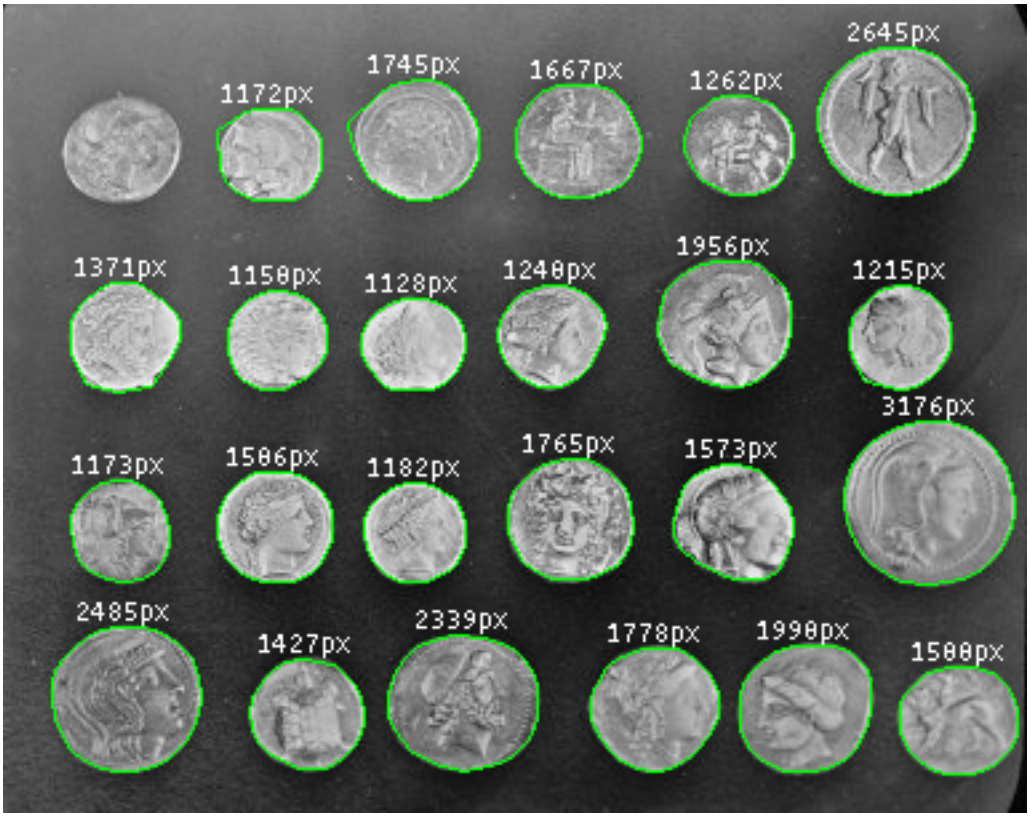
```
>>> from jicbioimage.segment import connected_components
>>> segmentation = connected_components(image, background=0)
>>> segmentation
```



The `jicbioimage.segment.connected_components()` function returns an instance of the `jicbioimage.segment.SegmentedImage` class, which provides access to segmented regions of interest as `jicbioimage.segment.Region` instances.

Finally, let us write a couple of functions to create an augmented reality image.

```
>>> import numpy as np
>>> from jicbioimage.illustrate import AnnotatedImage
>>> def text_position(region):
...     "Return x, y coordinates of text position."
...     ys, xs = region.index_arrays
...     y = np.min(ys) - 5
...     x = np.mean(xs, dtype=int)
...     return (y, x)
...
>>> def augment_image(image, segmentation):
...     "Return an augmented image."
...     augmented = AnnotatedImage.from_grayscale(image)
...     for i in segmentation.identifiers:
...         region = segmentation.region_by_identifier(i)
...         if region.area > 300 and region.area < 5000:
...             augmented.mask_region(region.convex_hull.border)
...             pos = text_position(region.convex_hull)
...             text = "{}px".format(region.convex_hull.area)
...             augmented.text_at(text, pos, center=True, antialias=False)
...     return augmented
...
>>> augmented = augment_image(array, segmentation)
>>> augmented
```


2645px
1745px
1667px
1172px
1262px
1956px
1371px
1150px
1128px
1240px
1215px
3176px
1173px
1506px
1182px
1765px
1573px
2485px
1427px
2339px
1778px
1990px
1500px

CHAPTER 4

Working with images

4.1 Introduction

There are several types of images in the `jicbioimage.core.image` module. Raw data is contained in the `jicbioimage.core.image.Image` class. The `jicbioimage.core.image.ProxyImage` and `jicbioimage.core.image.MicroscopyImage` classes contain image meta data along with a reference to the raw image.

The `jicbioimage.core.image.Image` is a subclass of `numpy.ndarray`. In addition to the `numpy.ndarray` functionality the `jicbioimage.core.image.Image` class has specialised functionality for creating images, tracking the history of images and returning png/html representations of images.

4.2 Creating images

4.2.1 Using `numpy` to create images

There are several ways of creating images. One can use the functionality inherited from `numpy.ndarray`.

```
>>> from jicbioimage.core.image import Image
>>> Image((50,50))
<Image object at 0x..., dtype=uint8>
```

Warning: When creating an image in this fashion it will be filled with the noise of whatever was present in that piece of computer memory before the memory was allocated to the image.

A safer way to create an image is to first create a `numpy.ndarray` using `numpy.zeros()` or `numpy.ones()` and then cast it to the `jicbioimage.core.image.Image` type.

```
>>> import numpy as np
>>> np.zeros((50,50), dtype=np.uint8).view(Image)
<Image object at 0x..., dtype=uint8>
```

When creating an array in this fashion it's history creation attribute is empty.

```
>>> print(np.zeros((50, 50), dtype=np.uint8).view(Image).history.creation)
None
```

To assign a creation event to the image history one can use the `jicbioimage.core.image.Image.from_array()` class method.

```
>>> ar = np.zeros((50, 50), dtype=np.uint8)
>>> im = Image.from_array(ar)
>>> im.history.creation
'Created Image from array'
```

4.2.2 Creating images from file

Suppose that we wanted to create an `jicbioimage.core.image.Image` instance from the file `images/rgb_squares.png`.

```
>>> fpath = "images/rgb_squares.png"
```

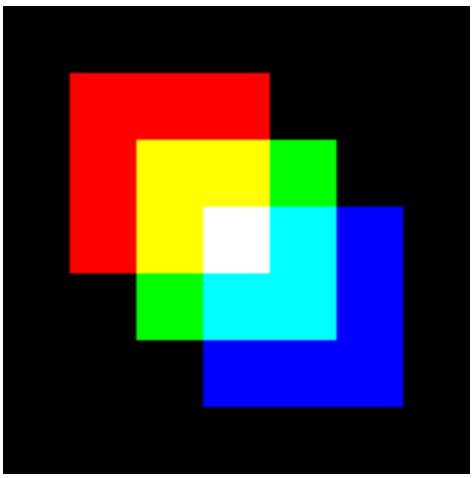
This can be achieved using the `jicbioimage.core.image.Image.from_file()` class method.

```
>>> im = Image.from_file(fpath)
```

4.3 Accessing png representations of an image

The `jicbioimage.core.image.Image.png()` function can be used to access the image as a PNG binary string. This function is used internally to implement the IPython integration, which allows images to be viewed directly in IPython qtconsole/notebook.

```
>>> im
```



4.4 Working with stacks of images

Many bioimages contain stacks of 2D images representing a 3D structure. The `jicbioimage.core.image.Image3D` class can be used to work with this type of data.

The `jicbioimage.core.image.Image3D` is a subclass of `numpy.ndarray`. To create an instance of a `jicbioimage.core.image.Image3D` from a numpy array and assign a creation event to the history of the 3D image one can use the `jicbioimage.core.image.Image3D.from_array` method.

To access such a stack from a `jicbioimage.core.image.MicroscopyCollection` one can use the `jicbioimage.core.image.MicroscopyCollection.zstack()` method.

```
>>> from jicbioimage.core.image import Image3D
>>> ar = np.zeros((50, 50, 50), dtype=np.uint8)
>>> im3d = Image3D.from_array(ar)
>>> im3d.history.creation
'Created Image3D from array'
```

It is possible to write and read an instance of `jicbioimage.core.image.Image3D` as a series of 2D images to and from a directory using the `jicbioimage.core.image.Image3D.to_directory()` method and `jicbioimage.core.image.Image3D.from_directory()` class method.

CHAPTER 5

Working with microscopy data

5.1 Introduction

One of the main driving forces behind the development of `jicbioimage` has been the need to simplify programmatic analysis of microscopy data.

Microscopy data differ from normal images in that they can be multidimensional. For example a microscopy image can consist of several z-slices, creating something like a 3D object, as well as several time points, creating something like a movie.

What this means in practise is that a microscopy datum is in reality a collection of 2D images. What `jicbioimage` tries to do is to provide easy access to the individual 2D images in the microscopy datum. This is achieved by unzipping the content of the microscopy datum into a backend, which acts as a type of cache of the individual 2D images.

However, microscopy data comes in a multitude of differing formats and it is not the intention that `jicbioimage` should understand these formats natively. Particularly as this is something that the Open Microscopy team already does through its BioFormats project.

In order to be able to process microscopy data `jicbioimage` therefore depends on the BioFormats command line tools. In particular the `bfconvert` tool, which is used to populate the backend.

For more information on how to install `jicbioimage` and the BioFormats command line tools please see the Installation notes.

5.2 Image collection classes

There are two image collection classes:

- `jicbioimage.core.image.ImageCollection`
- `jicbioimage.core.image.MicroscopyCollection`

These are used for managing access to the images stored within them. To this end the `jicbioimage.core.image.ImageCollection` class has got the functions below:

- `jicbioimage.core.image.ImageCollection.image()`
- `jicbioimage.core.image.ImageCollection.proxy_image()`

The `jicbioimage.core.image.MicroscopyCollection` class is more advanced in that individual images can be accessed by specifying the series, channel, zslice and timepoint of interest. For more information have a look at the API documentation of:

- `jicbioimage.core.image.MicroscopyCollection.image()`

- jicbioimage.core.image.MicroscopyCollection.proxy_image()
- jicbioimage.core.image.MicroscopyCollection.zstack_proxy_iterator()
- jicbioimage.core.image.MicroscopyCollection.zstack_array()

5.3 Obtaining image collections

One can obtain a basic jicbioimage.core.image.ImageCollection by loading a multipage TIFF file into a jicbioimage.core.io.DataManager. Let us therefore create a jicbioimage.core.io.DataManager.

```
>>> from jicbioimage.core.io import DataManager
>>> data_manager = DataManager()
```

Into which we can load the sample `multipage.tif` file.

```
>>> multipagetiff_fpath = "./tests/data/multipage.tif"
```

The jicbioimage.core.io.DataManager.load() function returns the image collection.

```
>>> image_collection = data_manager.load(multipagetiff_fpath)
>>> type(image_collection)
<class 'jicbioimage.core.image.ImageCollection'>
```

Which contains a number of jicbioimage.core.image.ProxyImage instances.

```
>>> image_collection
[<ProxyImage object at ...>,
 <ProxyImage object at ...>,
 <ProxyImage object at ...>]
```

5.4 Accessing data from microscopy collections

Suppose instead that we had a microscopy file. Here we will use the z-series.ome.tif file.

```
>>> zseries_fpath = "z-series.ome.tif"
```

Let us now load a microscopy file instead.

```
>>> microscopy_collection = data_manager.load(zseries_fpath)
>>> type(microscopy_collection)
<class 'jicbioimage.core.image.MicroscopyCollection'>
>>> microscopy_collection
[<MicroscopyImage(s=0, c=0, z=0, t=0) object at ...>,
 <MicroscopyImage(s=0, c=0, z=1, t=0) object at ...>,
 <MicroscopyImage(s=0, c=0, z=2, t=0) object at ...>,
 <MicroscopyImage(s=0, c=0, z=3, t=0) object at ...>,
 <MicroscopyImage(s=0, c=0, z=4, t=0) object at ...>]
```

One can now use a variety of methods to access the underlying microscopy images. For example to access the third z-slice one could use the code snipped below.

```
>>> microscopy_collection.proxy_image(z=2)
<MicroscopyImage(s=0, c=0, z=2, t=0) object at ...>
```

Alternatively to access the raw underlying image data of the same z-slice one could use the code snippet below.

```
>>> microscopy_collection.image(z=2)
<Image object at 0x..., dtype=uint8>
```

Similarly one could loop over all the slices in the z-stack using the code snippet below.

```
>>> for i in microscopy_collection.zstack_proxy_iterator():
...     print(i)
...
<MicroscopyImage(s=0, c=0, z=0, t=0) object at ...>
<MicroscopyImage(s=0, c=0, z=1, t=0) object at ...>
<MicroscopyImage(s=0, c=0, z=2, t=0) object at ...>
<MicroscopyImage(s=0, c=0, z=3, t=0) object at ...>
<MicroscopyImage(s=0, c=0, z=4, t=0) object at ...>
```

One can also access the z-stack as a `numpy.ndarray`.

```
>>> microscopy_collection.zstack_array()
array([[[ 0,  0,  0,  0,  0],
        [ 0,  0,  0,  0,  0],
        [ 0,  0,  0,  0,  0],
        ...
        [96, 96, 96, 96, 96],
        [96, 96, 96, 96, 96],
        [96, 96, 96, 96, 96]]], dtype=uint8)
```

However, it is often more convenient to access the z-stack as a `jicbioimage.core.image.Image3D` using the `jicbioimage.core.image.MicroscopyCollection.zstack()` method.

```
>>> microscopy_collection.zstack()
<Image3D object at 0x..., dtype=uint8>
```

CHAPTER 6

Working with transformations

6.1 Introduction

In image analysis one commonly wants to transform images. When putting an image through several transforms it can be really useful to save out the intermediate images to disk. Creating a visual audit track of the image processing.

To make it painless to set-up such an audit trail `jicbioimage` provides the function decorator `jicbioimage.core.transform.transformation()`. When applied to a transformation function the decorator adds both "autowriting" of the tranformed image as well as a log in the history of the image.

6.2 Pre-built transformations

The `jicbioimage.transform` package constains a number of standard image transformations that have had the `jicbioimage.core.transform.transformation()` function decorator applied to them.

For more information see http://jicbioimage.readthedocs.org/projects/jicbioimagetransform.

6.3 Creating a custom transform

Suppose that we wanted to create a transformation to invert our image. We can achieve this by importing the `jicbioimage.core.transform.transformation()` decorator.

```
>>> import numpy as np
>>> from jicbioimage.core.transform import transformation
>>> @transformation
... def invert(image):
...     """Return an inverted image."""
...     maximum = np.iinfo(image.dtype).max
...     maximum_array = np.ones(image.shape, dtype=image.dtype) * maximum
...     return maximum_array - image
...
```

Let us create an image to apply our tranformation to.

```
>>> from jicbioimage.core.image import Image
>>> ar = np.zeros((3,3), dtype=np.uint8)
>>> im = Image.from_array(ar)
```

We can now apply the transformation to our image.

```
>>> invert(im)
<Image object at 0x..., dtype=uint8>
```

6.4 Specifying `dtype` contracts

Sometimes one want to be able to ensure that the input/output image(s) are of a particular `dtype`. This can be achieved using the function decorator `jicbioimage.core.util.array.dtype_contract()`.

```
>>> from jicbioimage.core.util.array import dtype_contract
>>> @transformation
... @dtype_contract(input_dtype=bool, output_dtype=bool)
... def bool_invert(image):
...     """Return an inverted image."""
...     return np.logical_not(image)
...
```

If we try to apply this transform to an image of the wrong `dtype` we get an informative error message.

```
>>> bool_invert(im)
Traceback (most recent call last):
...
TypeError: Invalid dtype uint8. Allowed dtype(s): [<type 'bool'>]
```

6.5 Customising the behaviour of the visual audit trail

By default the audit trail images are written to the working directory. The location can be customised using `jicbioimage.core.io.AutoName.directory` attribute.

The generation of the audit trail images can be turned off by setting `jicbioimage.core.io.AutoWrite.on` attribute to `False`.

CHAPTER 7

IPython integration

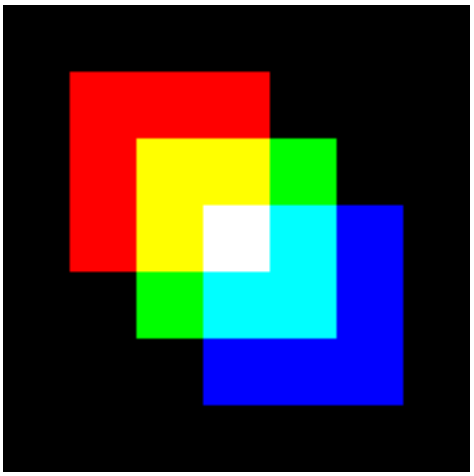
The image classes in `jicbioimage` have been designed to integrate with IPython qtconsole/notebook in that the image can be displayed directly in the console/notebook.

To illustrate the behaviour let us create a simple RGB image with some coloured squares.

```
>>> import numpy as np
>>> from jicbioimage.core.image import Image
>>>
>>> # Create the initial black image.
>>> ar = np.zeros((175, 175, 3), dtype=np.uint8)
>>> im = Image.from_array(ar)
>>>
>>> # Add full intensity to R, G, B channels at offset squares.
>>> im[25:100:, 25:100, 0] = 255
>>> im[50:125, 50:125, 1] = 255
>>> im[75:150, 75:150, 2] = 255
```

To display the image in the IPython qtconsole/notebook one simply needs access it.

```
>>> im
```



The behaviour works in IPython qtconsole/notebook with the classes listed below.

- `jicbioimage.core.image.Image`
- `jicbioimage.core.image.ProxyImage`
- `jicbioimage.core.image.MicroscopyImage`

- jicbioimage.segment.SegmentedImage
- jicbioimage.illustrate.Canvas
- jicbioimage.illustrate.AnnotatedImage

Furthermore the collection classes listed below will display summary information and thumbnails of all the images in the collection in IPython notebook.

- jicbioimage.core.image.ImageCollection
- jicbioimage.core.image.MicroscopyCollection

CHAPTER 8

API documentation

- jicbioimage.core.image
- jicbioimage.core.io
- jicbioimage.core.transform
- jicbioimage.core.util.array
- jicbioimage.core.util.color
- jicbioimage.transform
- jicbioimage.segment
- jicbioimage.illustrate