pyifx

Aug 30, 2019

Contents:

CHAPTER 1

Getting Started

1.1 Prerequisites

pip is required to install this library. If you do not have pip, refer to this guide regarding installation instructions.

1.1.1 Dependencies

The current dependencies of the library include:

- NumPy
- Imageio

However, these do not need to be installed beforehand. Installing the library will also install any dependencies it has if they are not found.

1.2 Installation

To install the library, use the command below:

```
pip install pyifx
```

This will install the library and its dependencies (if needed).

1.3 Importing the Library

To start using the library, import it into a python file.

```
"""This is a python file."""
import pyifx
```

Once the library is installed, it is ready for use.

CHAPTER 2

Image Classes

2.1 What are They?

All of the functions in this library are based around the use of Pyifx Image classes. They allow you to store important information about an image in Python while also providing useful functions relating to their properties.

2.2 Pyifx Image

The PyifxImage class allows for images to be read, modified, and written in combination with functions provided by the library. This class can either be instantiated through the use of an input and outputh path, or by providing given image data as long as it is in the form of a NumPy ndarray.

Below is an example of what creating instances of the PyifxImage class would look like.

```
>>> import pyifx
>>> image = pyifx.misc.PyifxImage(input_path="path/to/img.png", output_path="path/to/
↪new_img.png")
>>> image_from_data = pyifx.misc.PyifxImage(input_path=None, output_path="path/to/new_
↪img.png", img=image_data)
```

Once the class is instantiated, it reads the image located at the specified input path and converts it into a NumPy ndarray. This array can be easily manipulated & worked with to manipulate the represented image.

```
>>> image.get_image()
>>> array([[[174, 173, 213],
[174, 173, 213],
[174, 173, 213],
...,
[188, 183, 224],
[188, 183, 224],
[188, 183, 224]],
```

(continues on next page)

(continued from previous page)

```
[[174, 173, 213],
 [174, 173, 213],
 [174, 173, 213],
 ...,
 [188, 183, 224],
 [188, 183, 224],
 [188, 183, 224]],

[[174, 173, 213],
 [174, 173, 213],
 [174, 173, 213],
 ...,
 [188, 183, 224],
 [188, 183, 224],
 [188, 183, 224]],

...,

[[ 94, 110, 135],
 [ 94, 110, 135],
 [ 93, 109, 134],
 ...,
 [ 65, 107, 147],
 [ 65, 107, 147],
 [ 65, 107, 147]],

[[ 95, 111, 136],
 [ 94, 110, 135],
 [ 93, 109, 134],
 ...,
 [ 65, 107, 147],
 [ 66, 108, 148],
 [ 66, 108, 148]],

[[ 96, 112, 137],
 [ 95, 111, 136],
 [ 92, 108, 133],
 ...,
 [ 66, 108, 148],
 [ 66, 108, 148],
 [ 67, 109, 149]]], dtype=uint8)
```

The array is 3-dimensional, with the first dimension representing each row, the second for each pixel, and the third for each channel. The dimensions of the image can be viewed by accessing the shape property of the array.

```
>>> image.get_image().shape
>>> (1080, 1920, 3)
```

The reason the height comes before width is due to the fact that the first number represents the **number** of rows, which makes up the height of the image due to them being stacked on top of each other. The same goes for the width of the image, as well as the image channels.

The methods of this class include:

```
pyifx.misc.PyifxImage.refresh_image()
pyifx.misc.PyifxImage.get_input_path()
pyifx.misc.PyifxImage.set_input_path()
```

(continues on next page)

(continued from previous page)

```
pyifx.misc.PyifxImage.get_output_path()
pyifx.misc.PyifxImage.set_output_path()
pyifx.misc.PyifxImage.get_image()
pyifx.misc.PyifxImage.set_image()
```

Note: A full list & description of parameters & methods in the PyifxImage class can be found here or by visiting the library contents page.

2.3 Image Volume

The ImageVolume class is a tool used to create and collect PyifxImage instances for a large number of images. Instead of creating these images manually, the class will generate a list of PyifxImage instances based on a provided input directory. The generation method can also be tweaked through adjusting certain parameters when creating instances of the class (ex. Whether to include images from subdirectories.)

On instantiating this class, a 'volume' of images will be created based on the specified arguments. Provided below is an example of what using the class might look like.

```
>>> import pyifx
>>> volume = pyifx.misc.ImageVolume(input_path="lots/of/images/", output_path="lots/
↪of/images/modified/", prefix="_")
>>> print(volume.get_volume())
```

Running this file will show us what the generated list of images looks like.

```
>>> [<pyifx.misc.PyifxImage object at 0x0CC66E10>, <pyifx.misc.PyifxImage object at␣
↪0x0CC70030>, <pyifx.misc.PyifxImage object at 0x0CC66E50>]
```

Upon closer inspection, we can see what these images are based off of.

```
>>> image = volume.get_volume()[0]
>>> image.get_input_path()
>>> "lots/of/images/image_1.jpg"
```

And if we view the output path of the image, we can see where it leads to.

```
>>> image.get_output_path()
>>> "lots/of/images/modified/_image_1.jpg"
```

This is done for every image in the specified directory, and any subdirectories with images in it (if toggled).

The methods of this class include:

```
pyifx.misc.ImageVolume.volume_to_list()
pyifx.misc.ImageVolume.convert_dir_to_images()
pyifx.misc.ImageVolume.get_input_path()
pyifx.misc.ImageVolume.set_input_path()
pyifx.misc.ImageVolume.get_output_path()
pyifx.misc.ImageVolume.set_output_path()
pyifx.misc.ImageVolume.get_prefix()
pyifx.misc.ImageVolume.set_prefix()
pyifx.misc.ImageVolume.get_volume()
pyifx.misc.ImageVolume.set_volume()
```

Note: As stated before, a full list & description of parameters & methods in the ImageVolume class can be found here or by visiting the library contents page.

CHAPTER 3

Usage

3.1 Importing an Image

As stated in the previous section, most of pyifx's functionality is based around the use of image classes. There are three options to create data that is compatible with the Pyifx library.

3.1.1 PyifxImage

The PyifxImage class allows you to package data about an image into a class instance. This instance can then be passed to functions provided by the library to manipulate image data.

The snippet below shows an example of what creating an instance of this class would look like. You can view the full code example of this article here.

```
#demo_file.py
import pyifx

# Creating the image
image = pyifx.misc.PyifxImage(input_path="path/to/img.png", output_path="path/to/new_
↪img.png")
```

Note: More information about the PyifxImage class is available here.

3.1.2 ImageVolume

The ImageVolume class allows for the generation of a list of PyifxImage instances based on specified parameters. As with the PyifxImage class, instances of this class can be passed to library-provided functions as well.

Creating an ImageVolume instance is very similar to creating a PyifxImage instance. You can view the full code example of this article here.

```
# Creating the volume
volume = pyifx.misc.ImageVolume(input_path="lots/of/images/", output_path="lots/of/
↪images/modified/", prefix="_")
```

Note: More information about the ImageVolume class is available here.

3.1.3 PyifxImage list

Lists of PyifxImage instances can also be passed directly into pyifx functions. This can be used to import images from multiple directories, or if PyifxImage instances need to have properties that do not share any patterns or sequences.

Below is an example of what creating a PyifxImage instance list would look like. You can view the full code example of this article here.

```
#Creating the list
image_2 = pyifx.misc.PyifxImage(input_path="different/path/to/img.png", output_path=
↪"different/path/to/new_img.png")

image_list = [image, image_2]
```

3.2 Using Imported Images

3.2.1 Function Categories

Once an image is imported into an accepted class instance, it can be used by any of the main functions in the library. Functions are split into 4 categories based on their main purpose. These categories can be accessed based on the module names listed below:

```
pyifx.hsl
pyifx.graphics
pyifx.comp
pyifx.misc
```

Module name	Literal translation	Description
pyifx.hsl	HSL (Hue, Saturation, Light)	Functions in this section are focused around color & its manipulation. In simple terms, hue refers to the color itself, saturation to its intensity, and light to how bright or dark the color is.
pyifx.comp	Compositon	This section contains functions related to the manipulation of the properties of images. This includes its dimensions and file type among other properties.
pyifx.graphics	Graphics	This section focuses on the look and composition of images as a whole. Functions in this section mostly apply effects to images to change their look, such as blurs and pixelations.
pyifx.misc	Miscellaneous	Unlike the composition module, this section focuses on managing images instead of editing them. Functions and classes in this module include image classes and image import functions.

Note: A full list of functions is available here. To view functions contained in specific categories, visit the category's specific page mentioned in the table of contents.

3.2.2 Function Structure

pyifx functions accept any of the image classes mentioned in the *import section*. They return a new, modified instance of the same class or type as provided in the function. What is modified can vary based on what the function does. This is usually the image data; however, functions can also return modified input and output paths, prefixes (for ImageVolume instances), and other properties.

Below is an example of what using a pyifx function would look like.

```
brightened_image = pyifx.hsl.brighten(image, 50)
print(type(brightened_image))
```

If this file is run, we can see what the return value of this function would look like.

```
$ python demo_file.py
<class 'pyifx.misc.PyifxImage'>
```

The return value type always matches the image input type, regardless of the function.

```
brightened_list = pyifx.hsl.brighten(image_list, 50)
print(type(brightened_list))
```

```
$ python demo_file.py
<class 'list'>
```

3.3 Full Code Example

```
#demo_file.py
import pyifx

# Creating the image
image = pyifx.misc.PyifxImage(input_path="path/to/img.png", output_path="path/to/new_
↪img.png")

# Creating the volume
volume = pyifx.misc.ImageVolume(input_path="lots/of/images/", output_path="lots/of/
↪images/modified/", prefix="_")

#Creating the list
image_2 = pyifx.misc.PyifxImage(input_path="different/path/to/img.png", output_path=
↪"different/path/to/new_img.png")

image_list = [image, image_2]

brightened_image = pyifx.hsl.brighten(image, 50)
print(type(brightened_image))

brightened_list = pyifx.hsl.brighten(image_list, 50)
print(type(brightened_list))
```

CHAPTER 4

Contribution

Contribution is an essential step to improving and growing this project. If you want to contribute, please read this short guide detailing the steps you need to take to begin collaborating on this project.

4.1 Suggested Prerequisites

You can contribute in may ways besides helping write the underlying code; testers and technical writers are also very important to this project. Please read the list of suggested prerequisites before starting.

4.1.1 Library Contribution

This library is written in Python, so some fundamental understanding of Python can help you navigate the library. Additional libraries (both internal and external) used in the library include:

- NumPy
- Imageio
- OS
- Math

Understanding these libraries as a whole is not needed. However, understanding their purpose and what some of their main modules do can help when faced with code that uses them.

4.1.2 Documentation Contribution

Our doucmentation is written in Sphinx, a Python-based documentation tool. You can learn more about Sphinx by visiting their user guide.

For those looking to work on the structure of the documentation, some understanding of Sphinx and RST (reStructuredText) can be helpful. However, if you are more focused on the content of the documentation, then these tools

will not be of much use. No matter what you are focusing on, some understanding of these tools is still highly recommended.

4.2 How to Contribute

Any contributions to this project must be done through GitHub pull requests. If you aren't familiar with pull requests, please read this guide talking more about them.

All pull request descriptions must include:

- A short summary of what the change is doing
- Why the change is being made
- Any future expansions or plans to expand on the change (if any at all)

Any pull requests with insufficient data will be ignored.

Once a pull request is made, it will be reviewed. If the change is seen as beneficial or needed, it will be merged into the project.

4.3 Library Structure

4.3.1 Project Root

The root of the project contains files other than the library itself. Below is a table detailing the function of each of these files or directories.

File/Directory name	Purpose
docs	Contains all the documentation files for the project, including both build and source files
pyifx	The main library. Contains all of the source code and required package files. Any contributions to the library can be made here.
tests	Includes test materials & altered images.
tests_src	Contains the source files for tests.
.gitignore	Required for ignoring build files regarding version control.
LICENSE	The license for the project.
README.md	The README for the project.

4.3.2 Library Contents

The main library directory contains files that represent each module.

```
pyifx|
         hsl.py
         graphics.py
         comp.py
         misc.py
         INTERNAL.py
```

To reference a module in a python file (after importing the module), add the name of the module after `pyifx.`. For example, referencing the `hsl.py` file can be done by writing `pyifx.hsl`.

The library is split into 2 main parts; external and internal functions. External functions are found in the main 4 modules of the library (hsl, graphics, comp, misc) and are made to be used outside the library.

On the other hand, internal functions are made to be used by the library itself. It contains handlers and main algorithms for all of the library's features. These functions are located solely in the file `INTERNAL.py`, and can be referenced the same way as other modules.

Note: Currently, no documentation is provided for the INTERNAL module. It will be added in the coming weeks.

4.3.3 Function System

All external functions follow a specific system to handle inputs properly. The system follows a set of steps similar to the flow chart below:

```
External function -----------------> Function handler (INTERNAL) ----------------->␣
↪Function operation (INTERNAL)
 - Type checking                      - Value checking                              -
↪ Modifies image
                                      - Handles different image types               -
↪ Core image
```

- **External Function** - The function that is called by the user. This is where the user specifies the parameters of the function, and where the function arguments are checked for the correct type.
- **Function Handler** - This function checks the arguments themselves (ex. percent is between 0 and 100) instead of their type. It also handles the various types of image classes that can be entered and calls the appropriate functions in return.
- **Function Operation** - This is where the actual modification of the image happens. Unlike their handlers, most of these functions only accept PyifxImage instances, instead of the variety of types that the external functions accept.

Handler functions end with _handler, and operation functions end with _operation. Both handler and operation functions also begin with _, meaning they are internal and private. Any new features added to the library must follow this system.

4.4 Documentation Structure

Because the documentation is written in Sphinx & RST, it can mostly be edited using the same rules & syntax as any other project using the same tools. However, there are a few important exceptions to note.

Titles are underlined using =, subtitles use −, and sub-subtitles use ⋆. No overlining is required.

New functions are automatically documented as long as they have a docstring. Below is an example of the proper way to format a docstring.

```
""" detect_edges(img_paths, write=True)
        Takes image(s) and creates new images focusing on edges.

        :type img_paths: pyifx.misc.PyifxImage, pyifx.misc.ImageVolume, list
        :param img_paths: The image(s) to be manipulated.

        :type write: bool
        :param write: Whether to write the manipulated image(s).
```

(continues on next page)

(continued from previous page)

```
        :return: PyifxImage instance, ImageVolume instance, or list with elements of␣
↪type PyifxImage
        :rtype: pyifx.misc.PyifxImage, pyifx.misc.ImageVolume, list

"""
```

Classes need to have docstrings for each individual method. However, the class itself can have a docstring to document information about the class as well as its members. Below is an example of a class docstring.

```
""" A class used to create packages of images & their properties created for use with␣
↪the Pyifx library.

        :vartype input_path: str, NoneType
        :ivar input_path: The path to where the image is located. If the image does␣
↪not have an input path, it means that the instance is a result of combining two or␣
↪more images.

        :vartype output_path: str, NoneType
        :ivar output_path: The path to where edited images should be created. If the␣
↪image does not have an output path, it means the instance is used for read-only␣
↪purposes.

        :vartype image: numpy.ndarray, NoneType
        :ivar image: The image located at the input path in the form of a numpy n-
↪dimensional array. If the instance does not have an image property, it means that␣
↪the image had not been read.

"""
```

Here is a table of some of the common formatting tags used to reference certain parts of docstrings for both functions and classes.

Tag	Description
`:vartype MEMBER:`	The member type (for classes).
`:ivar MEMBER:`	The member description (for classes).
`:type PARAMETER:`	The parameter type (for functions).
`:param PARAMETER:`	The parameter description (for functions).
`:return:`	The return value description (for functions).
`:rtype:`	The return value type (for functions).

Although functions can be added to the API reference automatically, they still needed to be added to the 'Library Contents' page manually. If a new function has been approved, it needs to be added to the 'Library Contents' page manually. This page uses the Sphinx autosummary directive to add functions to the page. Below shows an example of what adding a new function would look like.

```
**pyifx.graphics**

.. autosummary ::

        pyifx.graphics.function_here
        pyifx.graphics.Class
        pyifx.graphics.new_function <---- New function here
```

If the module is new and must be added, write the module name (in the format of the module name above) and bold it.

Under that, add an autosummary directive and add the function to the list, making sure to follow the same format as the example above. Classes can be added in the same way.

4.5 Writing Tests

Writing tests is one of the most important parts of this project. Whether it is due to changing an existing feature or adding a new one, tests must be written in order to verify the validity of a change to the library. In order to keep tests organized, a few rules must be followed for writing them.

4.5.1 Location & Naming

As seen in the *project root table*, the **tests** directory contains test input & output files, while the **tests_src** directory is used to store test sources files. Any new tests **must** be written & saved in the tests_src directory.

Each test file is dedicated to an individual function. Functions must have individual test files and cannot be combined together. Files are named after the function as if they were being referenced from the root package.

```
tests_src |
        pyifx.hsl.brighten.py
        pyifx.hsl.darken.py
        ...
        pyifx.misc.combine.py
        ...
```

As seen above, the test file for the "brighten" function is named "pyifx.hsl.brighten.py", after its location in the package.

4.5.2 Test File Structure

Tests follow a specific structure in order to thoroughly cover every situation. Below is an example of what a test file would look like.

```
# pyifx.graphics.convolute_custom.py

# Import test materials
from test_vars import *

# Set output path(s)
set_paths("../tests/imgs/graphics/convolute_custom")

# Custom variables (optional)
sobel_horizontal_np = np.array([[-1,0,1], [-2,0,2], [-1,0,1]])
sobel_vertical = np.array([[-1,-2,-1], [0,0,0], [1,2,1]])
box_blur = np.array([[1/9, 1/9, 1/9], [1/9, 1/9, 1/9], [1/9, 1/9, 1/9]])

# Main tests
pyifx.graphics.convolute_custom(img1, sobel_horizontal_np)
pyifx.graphics.convolute_custom(img_list, sobel_vertical)
pyifx.graphics.convolute_custom(img_vol, box_blur)

# Error tests
call_error_test("pyifx.graphics.convolute_custom", ['s', sobel_horizontal_np])
```

(continues on next page)

(continued from previous page)

```
call_error_test("pyifx.graphics.convolute_custom", [img1, 's'])
call_error_test("pyifx.graphics.convolute_custom", [img1, sobel_horizontal_np, 's'])
```

- **Test materials**: Test materials, including variables & functions are located in the `test_vars.py` file, and can be imported using this statement.
- **Output path**: The directory to which any ouput files should be written. This path should be written in the format `../tests/imgs/*module*/*function_name*`.
- **Custom variables**: Any extra variables needed for the tests. These should only be included if required by the function.
- **Main tests**: Where the tests should be ran. The tests must be ran for all of the variables included in the *test materials*, & any other parameters should include a variety of values.
- **Error tests**: Where the error handling of the function is tested through the *call_error_test function*. Try to include all types of potential errors.

4.5.3 Test Materials

All materials needed for the test are provided in the `test_vars.py` file, which is located in the same directory as the rest of the tests. This file includes:

- Path-changing function (to help changing paths for individual variables)
- Error handler & catcher
- Variables (2x PyifxImage Instance, 1x ImageVolume, 1x Image List)

set_paths(*new_path*)

Changes the output path of test variables.

Parameters **new_path** (*str*) – The new output path.

call_error_test(*function*, *arguments*)

Calls a function with a provided list of arguments & catches any errors that arise. Prints message if caught successfully.

Parameters

- **function** (*str*) – The function to be called. Should be referenced from package (eg. pyifx.hsl.brighten). Does not include parentheses.
- **arguments** (*list*) – The arguments to be passed to the function. Should be contained in a list (in order of passing to the function).

Returns Boolean indicating if the error was caught.

Return type bool

CHAPTER 5

License

CHAPTER 6

Enquiries

If you have any questions or suggestions about the project, feel free to email me at jad@videolab.ae or jad.khalili123@gmail.com.

CHAPTER 7

API Reference

pyifx.hsl

pyifx.hsl.brighten(img_paths[, percent, write])	Takes image(s) and brightens them.
pyifx.hsl.darken(img_paths[, percent, write])	Takes image(s) and darkens them.
pyifx.hsl.color_overlay(img_paths, color[, ...])	Takes image(s) and applies a specified color over it/them.
pyifx.hsl.saturate(img_paths[, percent, write])	Takes image(s) and saturates them.
pyifx.hsl.desaturate(img_paths[, percent, write])	Takes image(s) and desaturates them.
pyifx.hsl.to_grayscale(img_paths[, write])	Takes image(s) and converts them to grayscale.

pyifx.graphics

pyifx.graphics.blur_gaussian(img_paths[, ...])	Takes images(s) and blurs them using a gaussian kernel based on a given radius.
pyifx.graphics.blur_mean(img_paths[, ...])	Takes images(s) and blurs them using a mean kernel based on a given radius.
pyifx.graphics.pixelate(img_paths[, factor, ...])	Takes image(s) and pixelates them based on a given factor.
pyifx.graphics.detect_edges(img_paths[, write])	Takes image(s) and creates new images focusing on edges.
pyifx.graphics.convolute_custom(img_paths, ...)	Takes image(s) and creates new images that are convoluted over using a given kernel.

pyifx.comp

pyifx.comp.resize(img_paths, new_size[, write])	Takes image(s) and converts them to a given size.
pyifx.comp.change_file_type(img_paths, new_type)	Takes image(s) and converts them to a given file type.

Continued on next page

Table 3 – continued from previous page

pyifx.comp.rewrite_file(img_paths)	Takes image(s) and writes them to an output destination based on their properties.

pyifx.misc

pyifx.misc.PyifxImage(input_path[, ...])	A class used to create packages of images & their properties created for use with the Pyifx library.
pyifx.misc.ImageVolume(input_path, output_path)	A class used to import images from a directory into Python, creating a list of PyifxImage instances.
pyifx.misc.combine(img1, img2, out_path[, write])	Combines the data of two PyifxImages, ImageVolumes, or ImageLists to form new PyifxImages.

CHAPTER 8

What is Pyifx?

Pyifx is an image processing, handling, & editing library meant to be used in Python. It provides users with the chance to edit and process images using Python code. The library provides over 20 main features to be used to edit images, including:

- Color overlay
- Blur
- Saturation
- Resize
- And much more.

A full list of available functions is available here.

The library was created by Jad Khalili, and first released in August 2019.

CHAPTER 9

Get Started

To install the library & begin using it, visit the Getting Started Guide.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index

C

S