Facelift Documentation

Release 0.2.1

Stephen Bunn <stephen@bunn.io>

Oct 30, 2020

CONTENTS

Several personal projects I've had in the past relied on some basic face feature detection either for face isolation, face state detection, or some kinds of perspective estimation. I found that there are plenty of resources for learning how to perform face detection in Python, but most of them suffer from a handful of the following issues:

1. Isn't easy to use right out of the box.
2. Doesn't provide face feature detection, just simple face detection.
3. Relies on older and no-longer maintained methods from `cv2` for face detection.
4. Is pretty greedy in terms of memory usage and scattered method calls.
5. Doesn't provide a selection of helpers to make face detection easier and quicker.
6. Requires that you write a whole bunch of boilerplate to get anything clean looking.
7. Or just my own personal disagreements with some of the code structure.

This project is my own attempt to provide decent face feature detection when you don't want to think too hard about it. We try to get as close as possible to a single `pip` install and still provide effective detection and recognition in Python. However, we do have several system dependencies that are necessary, see *System Requirements* for more details.

Below is a simple example of full face feature detection and rendering out to a standard OpenCV window using some of the features available in Facelift. **To get started using this package, please see the *Getting Started* guide.**

```
from facelift import FullFaceDetector, iter_stream_frames
from facelift.render import draw_line
from facelift.window import opencv_window

detector = FullFaceDetector()
with opencv_window() as window:
    for frame in iter_stream_frames():
        for face in detector.iter_faces(frame):
            for feature, points in face.landmarks.items():
                frame = draw_line(frame, points)

        window.render(frame)
```

CHAPTER

ONE

GETTING STARTED

Welcome to Facelift!
This page should hopefully provide you with enough information to get the `facelift` package installed so you can start detecting face features. If you run into any issues with installation, please create a Bug Report with details about your current operating system and package version and we can try to improve our setup documentation.

1.1 System Requirements

There are several required system requirements necessary for this package to work which we unfortunately cannot bundle in this package. The following sections will lead you through the installation of the necessary system requirements.

1.1.1 cmake

This tool is necessary as dlib needs to be built upon install.

Linux

Debian / Ubuntu

```
apt install cmake
```

MacOS

Homebrew

```
brew install cmake
```

Macports

```
port install cmake
```

Windows

Download the CMake installer and make sure to enable the setting to "Add CMake to the system PATH for all users" when installing. You may need to restart your shell depending on what terminal emulator you are using in Windows.

Make sure that you can run `cmake --version` in your shell without recieving a non-zero exit status code to verify your installation.

1.1.2 libmagic

This library helps us to determine the type of content we are attempting to process. We need this to be able to optimally determine how to consume the data for an arbitrary media file since OpenCV is pretty lacking in this area.

Linux

Debian / Ubuntu

```
apt install libmagic1
```

MacOS

Homebrew

```
brew install libmagic
```

Macports

```
port install file
```

Windows

We install python-magic-bin as a dependency if you are installing from a Windows environment. This package **should** contain working binaries for `libmagic` built for Windows. If you encounter unhandled errors using `libmagic` on Windows, please create an issue to let us know what you are experiencing.

1.2 Package Installation

Installing the package should be super duper simple as we utilize Python's setuptools.

```
$ poetry add facelift
$ # or if you're old school...
$ pip install facelift
```

Or you can build and install the package from the git repo.

```
$ git clone https://github.com/stephen-bunn/facelift.git
$ cd ./facelift
$ python setup.py install
```

Installing `opencv-python` *should* be quick for many environments as prebuilt packages are provided from PyPi. If you find that you are building OpenCV on installation, it's likely that you are installing an old version from `pythonhosted.org` which does **not** include prebuilt binaries. This will likely cause many issues with OpenCV not being built with proper support for GTK X11 support which is necessary for reading media and opening windows. If you run into this, try updating your local `pip` to the newest version (which should install the dependency from PyPi). Note that this dependency doesn't come prebuilt with any GPU support.

The `dlib` dependency will always need to be built when installing `facelift`. This requires that `cmake` is available on the system and doesn't build with any GPU support.

1.3 Model Installation

Due to PyPi's upload limits, we cannot bundle the associated landmark and ResNet models for face detection or face encoding. Similar to how other projects have dealt with this issue in the past, we have supplied a special module *_data* to programmatically fetch the necessary pre-trained models for using this package.

The *download_data()* function will attempt to fetch the models uploaded to the latest GitHub release.

```
from facelift._data import download_data
download_data()
```

If for some reason we mess up and forget to upload the models to the GitHub release, you can manually specify the release tag using the `release_tag` parameter. This will attempt to fetch the models from a very release instead of the very latest.

```
from facelift._data import download_data
download_data(release_tag="v0.1.0")
```

You can also see the basic download status written out to `stdout` by setting the `display_progress` parameter to `True`.

```
from facelift._data import download_data
download_data(display_progress=True)
```

I would prefer to be able to bundle the models along with the package since we are building a project revolving around **very specific** feature models and frameworks (rather than providing an open-ended framework for face detection). However, this is just something we need to do to satisfy PyPi.

Important: At the moment, the downloaded models will be placed in a `data` directory within the `facelift` package. This means that your system or virtual environment will contain the downloaded models. If you are interested in the absolute path that the downloaded models are being written to, you should set the `display_progress` flag to `True` as we write out where files are being stored.

1.4 GPU Support

To drastically speed up the processing and detection of face features we need to manually build both OpenCV and dlib for the machine's GPU. To do this we need to override the prebuilt CPU-only libraries included in the default installation of the package.

Tip: I'm going to try and get a guide together for building `opencv-python` and `dlib` with GPU support after a `v1.0.0` release as it is a secondary milestone for this project.

1.4.1 Building OpenCV

Todo: Need to write a guide for building OpenCV with GPU support for each platform.

1.4.2 Building Dlib

Todo: Need to write a guide for building dlib with GPU support for each platform.

CHAPTER

TWO

USAGE

Before we get started learning how to use the methods provided by *Facelift Package*, we have some basic terminology to define. The *types* module provides these following types/terms which we use throughout the package. We use these terms though most of our documentation, so make sure you take a peek at the responsibility of these names.

- *Frame*
  Defines a single matrix of pixels representing an image (or a single frame).
  Represented by a numpy.ndarray using the shape (Any, Any, 3) of type numpy.uint8.
  These frames are pulled out of some media or stream and is used as the source content to try and detect faces from.
- *Point*
  Describes an (x, y) coordinate relative to a specific frame.
  Represented by a numpy.ndarray of shape (2,) of type numpy.int64.
- *PointSequence*
  Describes a sequence of points that typically define a feature.
  Represented by a numpy.ndarray of shape (Any, 2) of type numpy.int64.
- *FaceFeature*
  An enum of available face features to detect (such as an eye or the nose).
  Represented by a *PointSequence*.
- *Face*
  Defines a detected face containing the landmarks and bounding frame of the face.
  Represented by a custom dataclasses.dataclass() using a dictionary of *FaceFeature* to *PointSequence* to describe the detected face features.
- *Encoding*
  Describes an encoded face frame that can later be used to recognize the same face.
  Represented by a numpy.ndarray of shape (128,) of type numpy.int64

2.1 Reading Frames

Likely you already have some kind of content you want to detect faces from. Whether that be a picture, a video, or your webcam, we need to be able to capture the frames from that media so we can use them for processing. These content types that we typically want to extract frames from are defined in *MediaType*.

class facelift.types.**MediaType**(*value*)
Enumeration of acceptable media types for processing.

IMAGE
Defines media that contains just a single frame to process.

VIDEO
Defines media that contains a known number of frames to process.

STREAM
Defines media that contains an unknown number of frames to process.

If processing a media file (such as an image or a video) these media types are automatically discovered from some magic methods available in the *magic* module. There, we attempt to make a best guess at what type of content you are attempting to capture frames from.

```
>>> from facelift.types import MediaType
>>> from facelift.magic import get_media_type
>>> media_type = get_media_type(Path("~/my-video.mp4"))
>>> assert media_type == MediaType.VIDEO
```

Actually opening and reading frames from content is typically performed using a mix of _ functions that use completel different syntax for each of these types of media. For most all use cases we really shouldn't care about the differences of how OpenCV opens, processes, and closes media. So we reduced the mental overhead of this process a bit and namespaced it within the *capture* module.

This module's overall purpose is to effeciently encapsulate the OpenCV calls necessary to capture the frames from the given media.

To do this we have exposed separate generator functions. One for handling written media files, and another for handling streamed frames. We made the decision to keep these generators separate as they have distinct features that would make a single generator function less explicit and intuitive.

2.1.1 Capturing Media Frames

To read frames from existing media files (either images or videos) you can utilize the *iter_media_frames()* generator to extract sequential frames. This function takes a `pathlib.Path` instance and will build the appropriate generator to capture and iterate over the available frames one at a time.

```
from facelift.capture import iter_media_frames
from facelift.types import Frame

for frame in iter_media_frames(Path("~/my-video.mp4")):
    assert isinstance(frame, Frame)
```

If you would like to loop over the available frames, the `loop` boolean flag can be set to `True`. This flag will seek to the starting frame automatically once all frames have been read essentially restarting the generator. This means that you will need to break out of the generator yourself as it will produce an infinite loop.

```
for frame in iter_media_frames(Path("~/my-video.mp4"), loop=True):
    assert isinstance(frame, Frame)
```

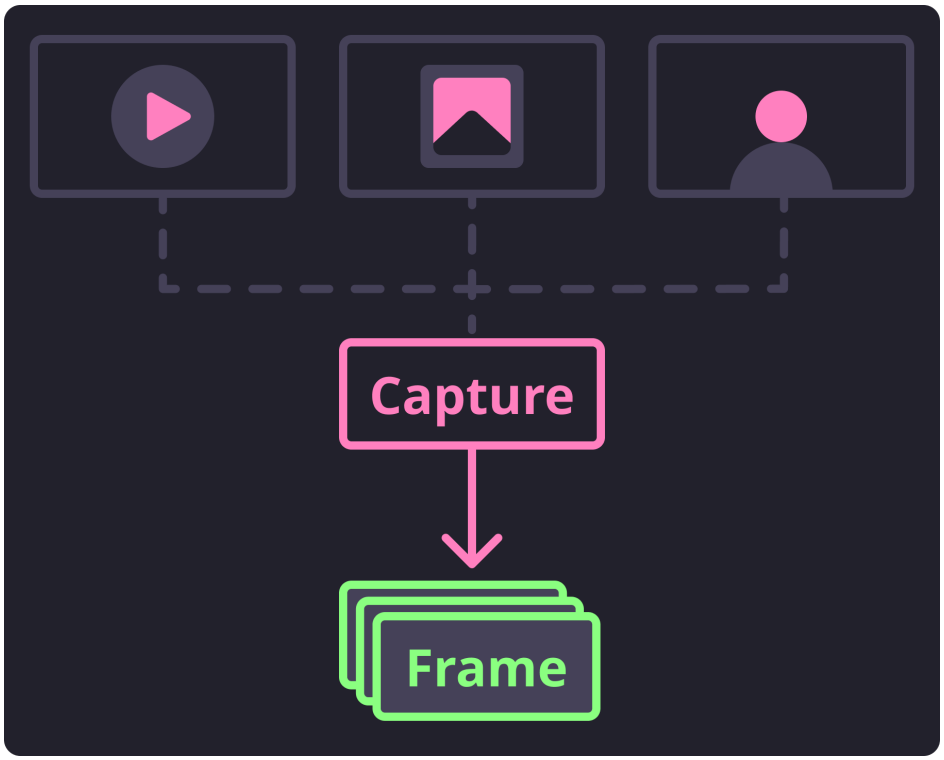

Fig. 1: Basic Capture Flow

2.1.2 Capturing Stream Frames

To read frames from a stream (such as a webcam) you can utilize the very similar *iter_stream_frames()* generator to extract the streaming frames. This function will scan for the first available active webcam to stream frames from.

```
from facelift.capture import iter_stream_frames
from facelift.types import Frame

for frame in iter_stream_frames():
    assert isinstance(frame, Frame)
```

If you happen to have 2 webcams available, you can pick what webcam to stream frames from by using the indexes (0-99). For example, if you wanted to stream frames from the second available webcam, simply pass in index `1` to the generator:

```
for frame in iter_stream_frames(1):
    assert isinstance(frame, Frame)
```

Important: When capturing streamed frames, this generator will not stop until the device stream is halted. Typically, when processing stream frames, you should build in a mechanism to break out of the capture loop when desireable.

In most of the below examples I will simply be raising `KeyboardInterrupt` to break out of this loop. You will likely want to add some kind of break conditional to this loop in your usage.

2.2 Rendering Frames

Now that we are reading frames in, we probably want to be able to preview what is going to be processed. OpenCV provides a *semi-decent* window utility that we take advantage of for our basic frame preview. If you want to display these frames in a more production-level application, I would recommend looking into using a canvas powered by OpenGL instead of relying on the hacky and inflexible solution provided by OpenCV.

> It is not within the scope of this project to provide an optimal canvas for displaying the frames read in through OpenCV. There are likely other projects out there that can display frames (numpy pixel arrays) or a transformed variant of this frame while taking advantage of the GPU.

Regardless, for our use cases we only want to be able to quickly and cheaply preview the frames we are processing. To help with this, we provide a *opencv_window* context manager that will create a temporary window that can be used for rendering these captured frames.

```
from facelift.capture import iter_media_frames
from facelift.window import opencv_window

with opencv_window() as window:
    for frame in iter_stream_frames():
        window.render(frame)
```

Here is a quick screen capture running the above example.

Note that since this window helper is a context manager, the window will destroy itself once there are no more frames to process and we break out of the frame generator. In the above example, I am simply raising a `KeyboardInterrupt` by pressing `Ctrl+C`, but you can be much more clever about it in your usage.

2.2.1 Customization

There are several available options that allow you to *slightly* tweak the created window. The options are fairly limited as we are just forwarding the desired tweaks to the creation of the window in OpenCV. *Don't expect much in terms of flexibility of customization for these windows.*

Window Title

By default, we throw a "Facelift" title on the created window.
You can override this by passing in a `title` to the context manager:

```
with opencv_window(title="My Window") as window:
    ...
```

This title will be used to also destroy the window as OpenCV naively destroys windows based on window titles. This isn't such a big issue as OpenCV (and in turn the *opencv_window* manager) doesn't allow mutation of a window title once the window is opened.

Window Style

OpenCV windows have several different style features they can pick and choose from. These features are defined in the *WindowStyle* object and can be joined together with the boolean `|` and passed through to the `style` parameter.

```
from facelift.window import opencv_window, WindowStyle

with opencv_window(style=WindowStyle.GUI_EXPANDED | WindowStyle.KEEP_RATIO) as window:
    ...
```

By default the window will use the *DEFAULT* window style which is a combination of some of other available window styles. If you actually need to use a custom window style, I encourage that you play around with these options yourself to see what works best for you.

Display Delay

The delay at which OpenCV attempts to render frames is another feature that can be controlled. This is fairly useful when you want to slow down the frames being rendered in the window rather than the speed at which frames are being read. This delay is defined in milliseconds as an integer and is defaulted to 1.

```
from facelift.capture import iter_stream_frames
from facelift.window import opencv_window

with opencv_window(delay=1000) as window:  # wait 1 second between displaying frames
    for frame in iter_stream_frames():
        window.render(frame)
```

Note that you can also handle do this yourself with a simple `time.sleep()` prior or post a *render()* call. That solution may be a better path forward if you are running into issues with the `delay` parameter.

Warning: This delay **must** be greater than 0. We have a validation step in the creation of the window to ensure that it is not initialized to 0. However, you can still get around this initial check by setting `delay` on the created window context instance. For example, you can *technically* do the following:

```
>>> from facelift.window import opencv_window
>>> with opencv_window(delay=1) as window:
...   window.delay = 0
```

This will very likely break the frame rendering as OpenCV will enter a waiting state with no refresh interval when the window delay is set to 0.

Display Step

Sometimes you want to pause on each frame to *essentially* prompt for user interaction when rendering frames. This feature is particularly useful when attempting to render single frames (such as those from images) as the generator will immediately exit and could exit the window context manager which will destroy the window.

For example, the following sample will immediately create a window and then quickly close it as the *iter_media_frames()* generator will immediately read the image and immediately exit the window's context manager:

```
with opencv_window() as window:
    for frame in iter_media_frames(Path("~/my-image.jpeg")):
        window.render(frame)
```

If you would like to force the window to await user input to render the next frame every time *render()* is called, you can use the `step` and `step_key` arguments.

```
with opencv_window(step=True, step_key=0x20) as window:
    ...
```

In the above example, since we have enabled `step` and defined the step key to be `0x20`, our window will wait for the user to press [Space] (ASCII 36 or 0x20) before rendering the next frame.

2.3 Transforming Frames

Before we get to actually detecting faces, it would benefit us to know what kind of bottlenecks we will hit and how we can avoid or reduce them.

The obvious bottleneck any kind of object detection is that the more pixels you have to process, the longer object detection takes. To reduce this we typically want to scale down large frames so that we don't waste so much time looking through all the available pixels. This scaling operation is provided as a transformation function *scale()*.

```
from facelift.capture import iter_stream_frames
from facelift.transform import scale

for frame in iter_stream_frames():
    assert frame.shape[0] == 128
    frame = scale(frame, 0.5)
    assert frame.shape[0] == 64
```

By scaling down the frame to a more reasonable size, feature detection will be able to perform much quicker as we have less pixels to run through. This is just one example of how we can reduce bottlenecks to benefit feature detection. However, there are many more transformations that we *might* need to do to benefit `dlib`'s frontal face detector.

For example, what if we are processing a video shot in portrait but we are reading in frames in landscape? We will probably need to rotate the frame to be in portrait mode so that the faces we are trying to detect are positioned top-down in the frame instead of left-right. We can also do this using a provided transformation *rotate()*.

Let's say we want to rotate these frames -90 degrees:

```
from facelift.capture import iter_stream_frames
from facelift.transform import rotate

for frame in iter_stream_frames():
    frame = rotate(frame, -90)
```

For a full list of the available transformations we supply, I recommend you look through the *transform* module's auto-built documentation.

The goal of this module is to provide the basic transformations that you may need to optimize face detection using our methods. You may run into a use case where you need something we do not provide in this module. In this case, you likely can find what you need already built into OpenCV.

2.3.1 Chaining Transforms

Most of the time you will end up with several necessary transformations to get the frame in a position that is optimal for face detection. In these cases, it's fairly straightforward to compose multiple transforms together through the following type of composition:

```
from facelift.capture import iter_stream_frames
from facelift.transform import scale, flip, rotate
from facelift.window import opencv_window

with opencv_window() as window:
    for frame in iter_stream_frames():
        frame = rotate(flip(scale(frame, 0.35), x_axis=True), 90)
        window.render(frame)
```

In this example, we are first scaling down the frame to 35%, flipping the frame on the x-axis, and the rotating it by +90 degrees. Potentially useful for large, inverted media files where faces are aligned left to right rather than top-down. Internally the frame is going through each transformation just as you would expect.

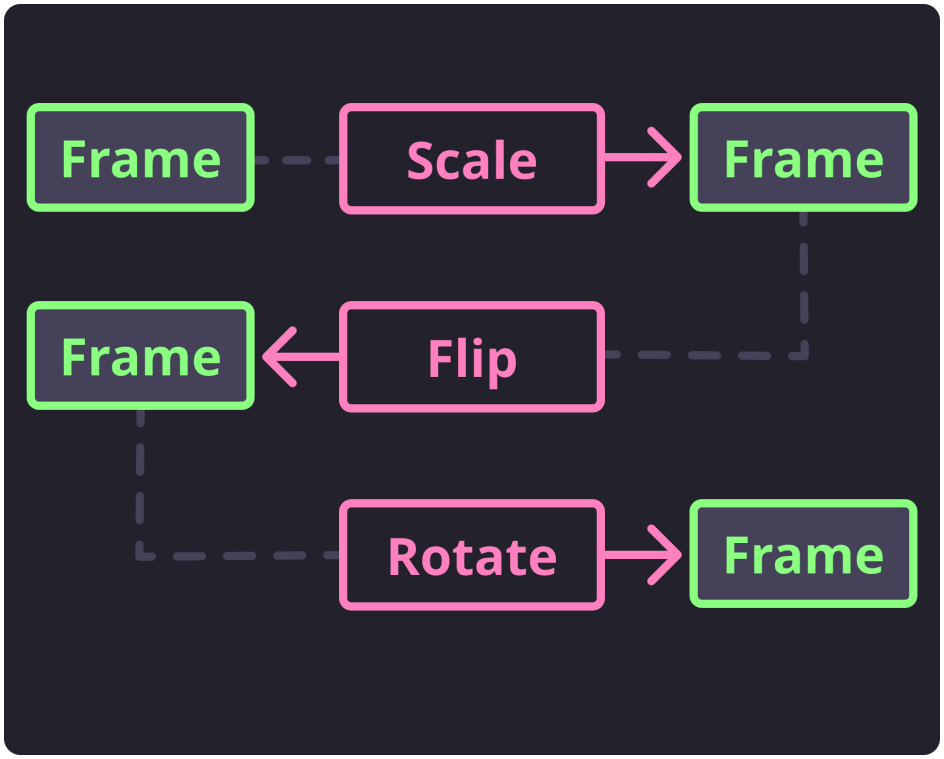

Fig. 2: Sample Transform Flow

This was just a quick overview of the concept of transforming frames before we attempt to detect face features. We will see more explicitly how transformations benefit feature detection in the next section.

2.4 Drawing on Frames

We've included a few drawing features are just quick and easy to use wrappers for some of the builtin drawing functionality OpenCV provides by default. These helper functions are found in the *render* module. The original reason we added these drawing helpers in was to make it easier to debug what is being detected.

As these drawing features are already common to OpenCV, we will give just a quick overview of what is available in this package. For a more full description of the available parameters and constants, you should just read through the *render* module auto-generated documentation.

2.4.1 Points

Drawing points is really simple. If you have a single point either as an `(x, y)` tuple or a *Point* that you want to render you can use *draw_point()*.

```
from facelift.render import draw_point
frame = numpy.zeros((512, 512, 3))
frame = draw_point(frame, (64, 64), size=4)
```

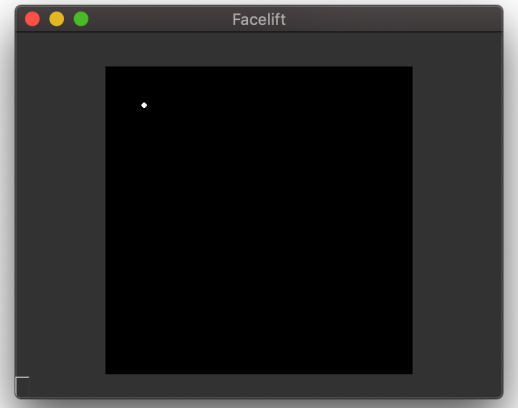

Fig. 3: Drawing a single point

We also provide a shorthand function for rendering either a list of `(x, y)` tuples or an *PointSequence* called *draw_points()*.

```
from facelift.render import draw_points
frame = numpy.zeros((512, 512, 3))
frame = draw_point(frame, [(64, 64), (128, 128), (256, 256)], size=4)
```

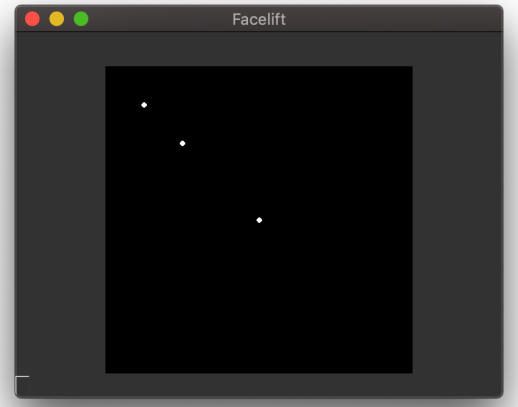

Fig. 4: Drawing multiple points

2.4.2 Lines

If you have a list of `(x, y)` tuples or an *`PointSequence`* and you wish to draw a connected line between the points, you can use the *`draw_line()`* function.

```
from facelift.render import draw_line
frame = numpy.zeros((512, 512, 3))
frame = draw_line(frame, [(64, 64), (128, 128), (256, 256)], thickness=4)
```

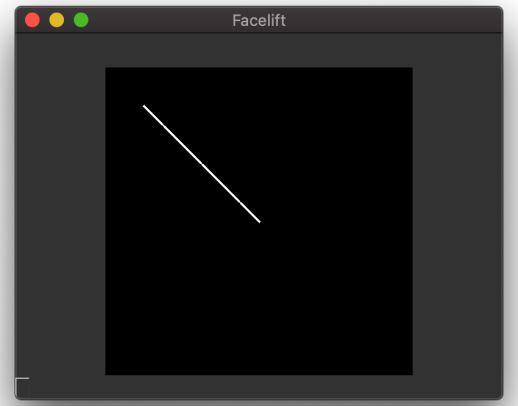

Fig. 5: Drawing a line

2.4.3 Shapes

Instead of having to use a combination of just points and lines to draw everything, we have a few other functions that provide basic shape drawing. These are still mostly all just wrappers around the default functionality that OpenCV provides.

Rectangles

Drawing rectangles is as simple as providing top-left and bottom-right points to draw the rectangle between.

```
from facelift.render import draw_points
frame = numpy.zeros((512, 512, 3))
frame = draw_point(frame, (64, 64), (256, 256), thickness=2)
```

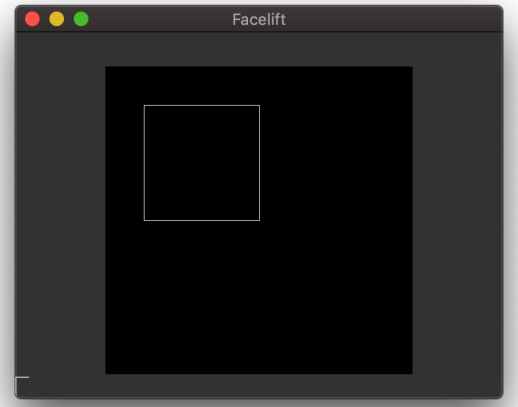

Fig. 6: Drawing a rectangle

Circles

Circles are just points with a non-negative thickness. So to draw a circle we can utilize the included *draw_point()* and supply *at-least* a thickness of `0`.

```
from facelift.render import draw_point
frame = numpy.zeros((512, 512, 3))
frame = draw_point(frame, (64, 64), size=32, thickness=0)
```

You will probably also want to adjust the `size` of the point as a small enough point will always appear filled rather than as a circle.

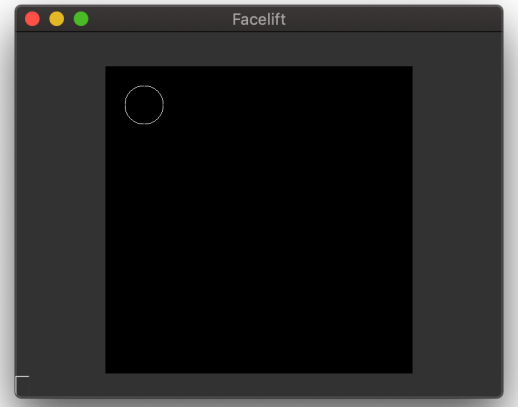

Fig. 7: Drawing a circle

Contours

If you want to use an *PointSequence* as the outline for a shape, you can use the *draw_contour()* function.

```
from facelift.render import draw_contour
frame = numpy.zeros((512, 512, 3))
frame = draw_contour(frame, [(64, 64), (128, 128), (256, 256), (64, 256)])
```

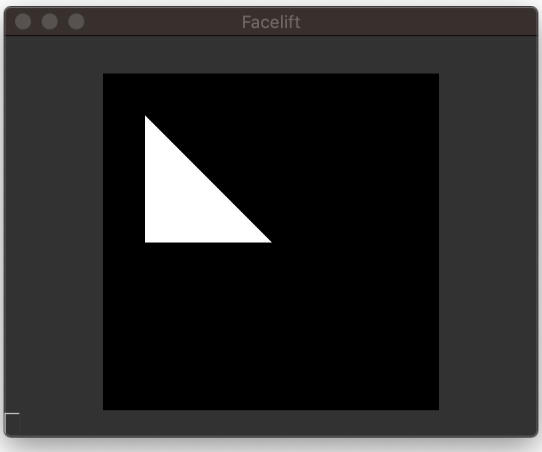

Fig. 8: Drawing a triangle

2.4.4 Text

Drawing text is a *bit* more complicated than the other helper functions. Rather than have to do some messy calls to determine width and height of specific fonts to render text in the appropriate location, we handle drawing text by first defining a bounding box for the text to be positioned in. Since it is **much** easier to place a rectangle, drawing text within that rectangle's bounds is much easier in turn. This is all handled by the *draw_text()* function.

In the below examples, we are drawing a red rectangle to visualize where the text lives within the defined text container. The defined container from the call to *draw_text()* will be invisible.

```
from facelift.render import draw_rectangle, draw_text
frame = numpy.zeros((512, 512, 3))
frame = draw_rectangle(frame, (64, 64), (448, 256), color=(0, 0, 255))
frame = draw_text(frame, "Hello, World!", (64, 64), (448, 256))
```

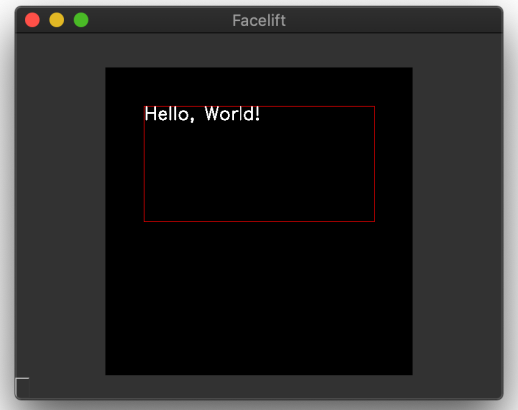

Fig. 9: Default aligned text

You can utilize the *Position* enumeration to position the text within this bounding box. For example, if we wanted to center the text we can set both the `x_position` and `y_position` to *CENTER*.

```
from facelift.render import draw_rectangle, draw_text, Position
frame = numpy.zeros((512, 512, 3))
draw_rectangle(frame, (64, 64), (448, 256), color=(0, 0, 255))
frame = draw_text(
    frame,
    "Hello, World!",
    (64, 64), (448, 256),
    x_position=Position.CENTER,
    y_position=Position.CENTER
)
```

Similarly you can set both to *END* to place the text at the lower left corner of the text container.

```
from facelift.render import draw_rectangle, draw_text, Position
frame = numpy.zeros((512, 512, 3))
draw_rectangle(frame, (64, 64), (448, 256), color=(0, 0, 255))
frame = draw_text(
    frame,
```

(continues on next page)

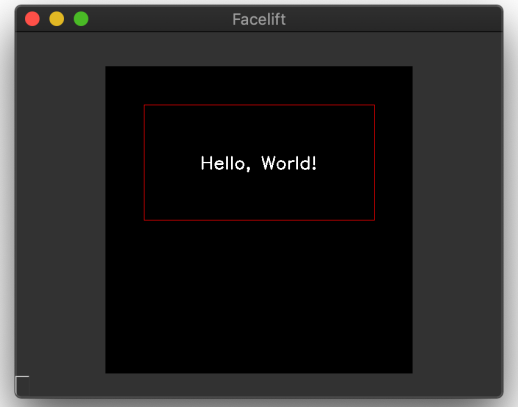

Fig. 10: Center aligned text

(continued from previous page)

```
    "Hello, World!",
    (64, 64), (448, 256),
    x_position=Position.END,
    y_position=Position.END
)
```

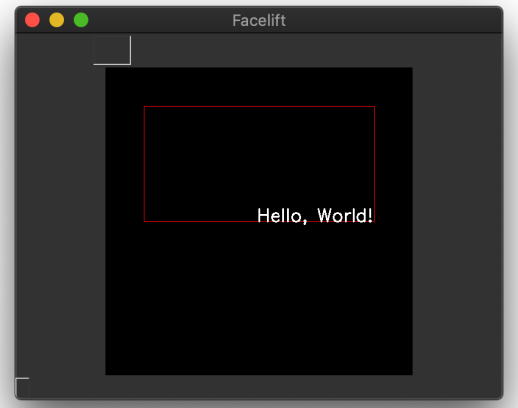

Fig. 11: End aligned text

Container Overflow

If you don't actually want to utilize the bounding box as a container, and instead want to use it as basically just a big reference to start OpenCV's default text drawing, you can set `allow_overflow` to `True`.

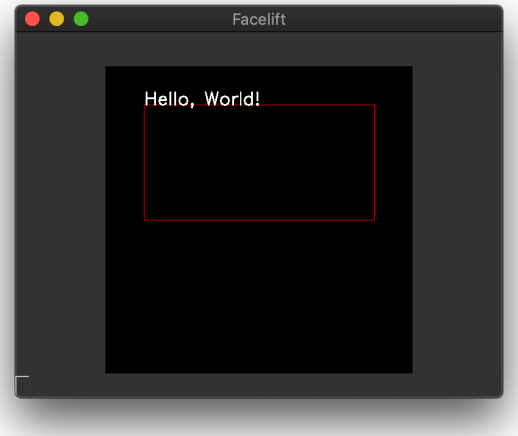

Fig. 12: Default aligned text with `allow_overflow=True`

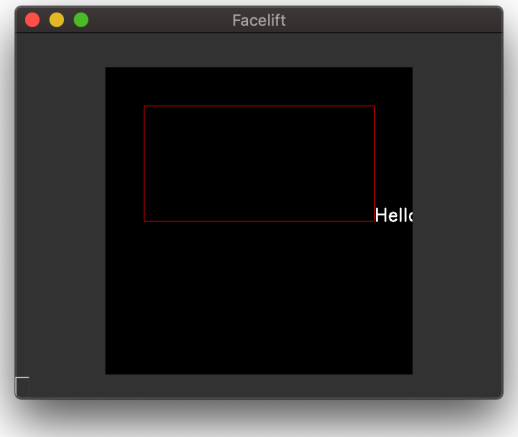

Fig. 13: End aligned text with `allow_overflow=True`

Warning: We are not being clever enough here to handle any kind of text wrapping for the bounding container that you define. This container is really only used to determine where to **start** drawing your text rather than keeping it all within a box.

If your text size is larger than the defined bounding container, it will overflow (likely on the x-axis).

2.5 Detecting Faces

Now onto the fun part. Face feature detection is powered by good ol' dlib. As part of this package, we have provided 3 pre-trained landmark models for face features each detecting various different face landmarks.

Tip: To learn how to acquire these models, please see the *Model Installation* documentation.

The face features (interchangeably termed landmarks) we are able to detect are classified in the *FaceFeature* enumeration.

The landmark models are programmatically provided through the following *BaseLandmarkDetector* subclasses:

- *BasicFaceDetector*
- *PartialFaceDetector*
- *FullFaceDetector*

Although each of these detectors operates *essentially* the same way, they produce different results and have various caveats that you would benefit from knowing. We will be covering the ins and outs of each of these detectors in the following related sections.

But first, a few helpful details on how subclasses of *BaseLandmarkDetector* work. Each subclass will contain a reference to a file location where the trained landmark model exists. Upon instantiation of the subclass, that model will be open and read into memory **which can take longer than a second in some cases** (given the size of the model).

```
from facelift.detect import BasicFaceDetector
BasicFaceDetector.model_filepath  # filepath to the related trained landmark model
detector = BasicFaceDetector()    # trained model is opened and loaded into memory
```

Each instance of these subclass comes with a helpful little generator that will produce instances of *Face* for a given *Frame*. This generator is called *iter_faces()* and will utilize the loaded model to handle feature detection automagically .

```
from facelift.types import Face
from facelift.capture import iter_stream_frames
from facelift.detect import BasicFaceDetector

detector = BasicFaceDetector()
for frame in iter_stream_frames():
    for face in detector.iter_faces(frame):
        assert isinstance(face, Face)
```

Each generated face represents a single set of face landmarks detected from the given frame. The accuracy of this detection is wholely the responsibility of the trained model (although you can typically benefit it by transforming the frame into an optimal state before attempting to perform detection).

If you are finding that the face landmark models we install are not as accurate as you require, you should look further into training your own landmark models for dlib. **Note that this is not a trivial task.**

Tip: The *iter_faces()* generator comes with a parameter called `upsample` that is defaulted to `0`. If you are having to detect faces from **really** small frames, setting this parameter to a positive value will attempt to optimally upsample the frame using dlib's builtin utilities.

```
for frame in iter_stream_frames():
    for face in detector.iter_faces(frame, upsample=2):
        assert isinstance(face, Face)
```

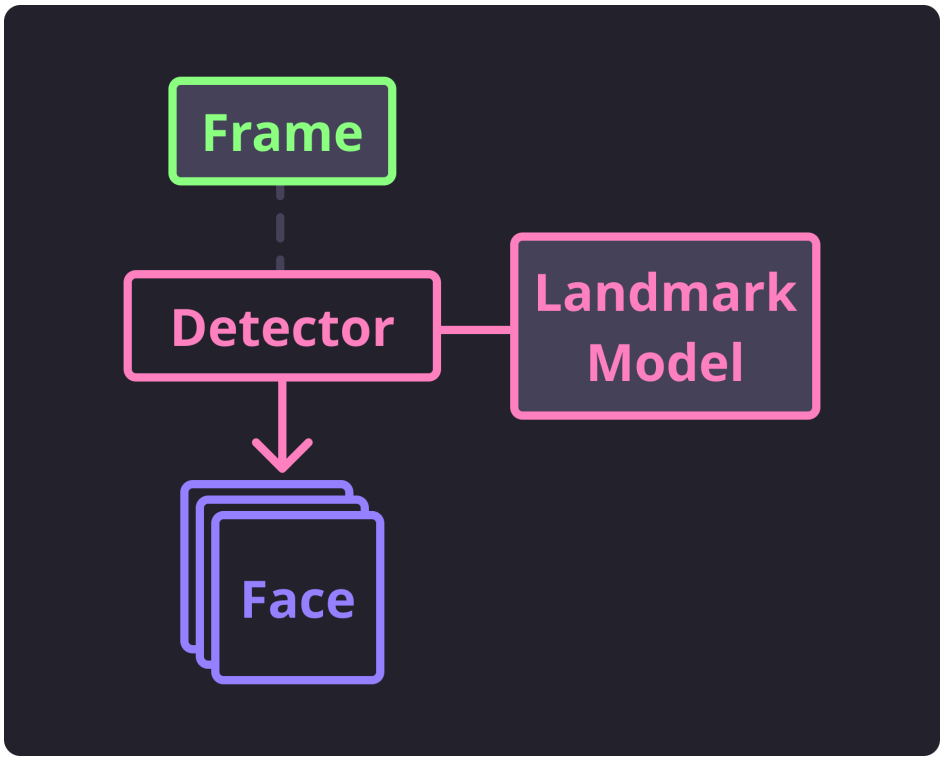

Fig. 14: Detect Flow

Be cautious about using this feature with large frames as it will drastically increase the amount of time that is necessary to detect faces. I would recommend avoiding using this feature when processing multiple frames (either from videos or streams).

2.5.1 Basic Face Detection

The basic face detector is the lightest weight detector and likely should be used for when you need to simply detect faces or recognize faces. Faces detected with this detector contain a single point for three face features:

- *LEFT_EYE* - A single point on the outside of the left eye
- *RIGHT_EYE* - A single point on the outside of the right eye
- *NOSE* - A single point right below the nose

Checkout the following recording of the below script for a better understanding of what points are detected.

```
from facelift.capture import iter_stream_frames
from facelift.detect import BasicFaceDetector
from facelift.window import opencv_window
from facelift.render import draw_points

detector = BasicFaceDetector()
with opencv_window() as window:
    for frame in iter_stream_frames():
        for face in detector.iter_faces(frame):
            for _, points in face.landmarks.items():
```

(continues on next page)

(continued from previous page)

```
                # big colorful points so you can see what's going on
                frame = draw_points(frame, points, size=3, color=(0, 255, 0))

        window.render(frame)
```

Because the features we are detecting are just single points, we really can't do much to determine a face's state (such as if eyes are opened or closed). However, we can determine where a face is placed and what angle the face is tilted within the frame. This is particularly helpful as we usually want to extract normalized frames with a properly positioned face for more accurate face recognition.

As an example of this, we have included the *helpers* module with some examples of basic face positioning math to extract frames where the face is always centered and angled correctly. The helper function that provides this appropriate face positioning functionality is *get_normalized_frame()*. You can use it by simply passing both the starting frame and a detected face:

```
from facelift.capture import iter_stream_frames
from facelift.detect import BasicFaceDetector
from facelift.window import opencv_window
from facelift.helpers import get_normalized_frame

detector = BasicFaceDetector()
with opencv_window() as window:
    for frame in iter_stream_frames():
        for face in detector.iter_faces(frame):
            frame = get_normalized_frame(frame, face)

        window.render(frame)
```

Overall, the *BasicFaceDetector* is useful for quick face detection where your only desire is to extract a face from a frame.

2.5.2 Partial Face Detection

The *PartialFaceDetector* uses the heaviest of the three landmark models (likely since it was trained the most rigorously of the three). This detector detects all face features **except** for the *FOREHEAD* feature. Each detected feature is a `PointSequence` and can be used to render the outline of the detected face.

```
from facelift.capture import iter_stream_frames
from facelift.detect import PartialFaceDetector
from facelift.window import opencv_window
from facelift.render import draw_points

detector = PartialFaceDetector()
with opencv_window() as window:
    for frame in iter_stream_frames():
        for face in detector.iter_faces(frame):
            for _, points in face.landmarks.items():
                frame = draw_points(frame, points, color=(0, 255, 0))

        window.render(frame)
```

Because this detector is discovering multiple points for a single face feature, we can use these points to actually build a pretty good representation of the face.

2.5.3 Full Face Detection

The *FullFaceDetector* is a third-party trained model that includes detection of all *FaceFeature* features. With the addition of the *FOREHEAD* feature, we can also include the curvature and angle of the forehead.

```
from facelift.capture import iter_stream_frames
from facelift.detect import FullFaceDetector
from facelift.window import opencv_window
from facelift.render import draw_points

detector = FullFaceDetector()
with opencv_window() as window:
    for frame in iter_stream_frames():
        for face in detector.iter_faces(frame):
            for _, points in face.landmarks.items():
                frame = draw_points(frame, points, color=(0, 255, 0))

        window.render(frame)
```

This model is not as heavily trained as the *PartialFaceDetector* so you may see some inconsistencies between the two detectors. Regardless, with the inclusion of the *FOREHEAD* feature, you get another dimension to work with that may be valuable for your use case.

2.6 Recognizing Faces

Recognition is performed by producing an *Encoding* for a detected face. This encoding is just an array of dimensions that *should* be pretty unique for that person's face. The encoding itself is produced by yet another pre-trained model produced by dlib. This model is a ResNet model trained for producing identifiers for images of faces. There are other trained models for producing identifiers for detected faces, however we are only bundling the one produced and used by dlib.

2.6.1 Encoding Faces

Similar to how we handle face detection, we also provide a *BasicFaceEncoder* from the *encode* module. This encoder provides a method *get_encoding()* which will take a given frame and a face detected within that frame to produce an *Encoding* for the face.

You can quickly get a face's encoding from a script similar to the following:

```
from pathlib import Path
from facelift.encode import BasicFaceEncoder
from facelift.detect import BasicFaceDetector
from facelift.capture import iter_media_frames

detector = BasicFaceDetector()
encoder = BasicFaceEncoder()

frame = next(iter_media_frames(Path("~/my-profile-picture.jpeg")))
face = next(detector.iter_faces(frame))
face_encoding = encoder.get_encoding(frame, face)
```

You will note that the name *BasicFaceEncoder* is very similar to *BasicFaceDetector*. This is to hopefully encourage developer's intuition to use these two classes together when performing face recognition.

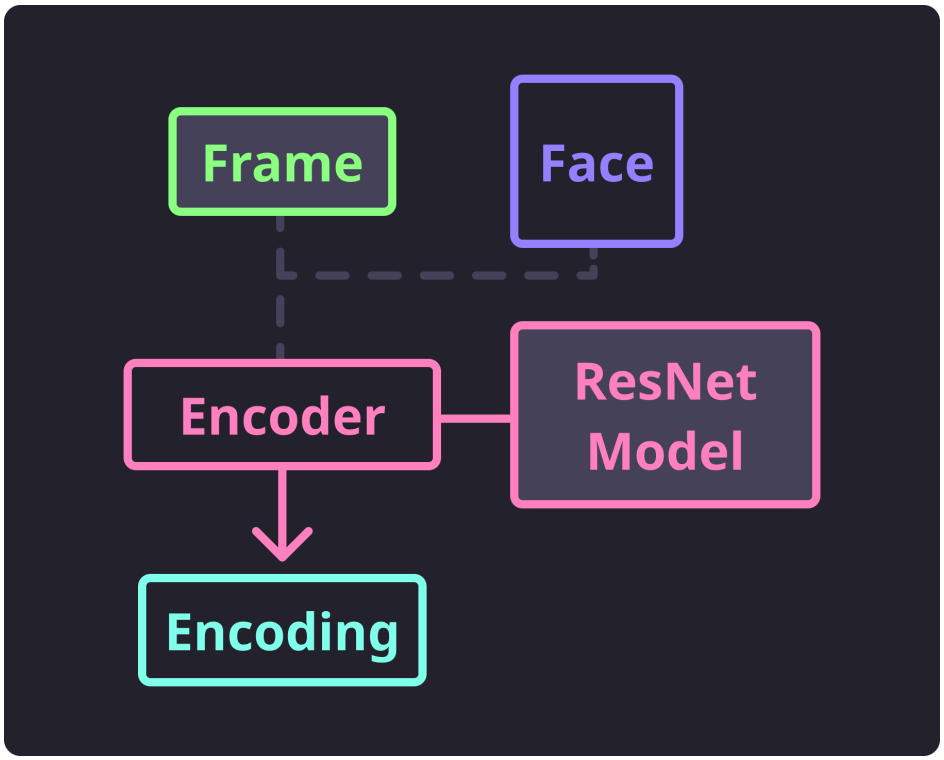

Fig. 15: Encode Flow

Important: Face recognition with the *BasicFaceEncoder* **will not work** from faces detected using the *FullFaceDetector*.

Although you can use faces detected from both the *BasicFaceDetector* and *PartialFaceDetector* to get encodings from this *BasicFaceEncoder*, **you should always prefer using lighter weight detector to avoid slowdown.**

This module **does not** provide any kind of features for storing these produced encodings; that is completely up to the implementation you are building. You will need to find a way to store the produced encodings associated to an identifier (such as the persons name). For example, you could really simply store the encoding directly associated with the person's name by using a dictionary such as the following:

```
face_encoding = encoder.get_encoding(frame, face)
face_storage = {
    "Stephen Bunn": [face_encoding]
}
```

Remember that each encoding is an instance of a `numpy.ndarray` which isn't immediately JSON serializable. However, they can be converted to more common types or can be stored using `pickle` or something more advanced.

2.6.2 Scoring Encodings

We haven't yet actually performed any recognition yet. But now that we have some registered encodings, we can start taking newly detected faces and score them against our known face encodings to get a good idea whose face we are detecting. This scoring is provided by the *score_encoding()* method which takes an unknown face encoding and a list of known faces for a **single** person to see how similar they are. The closer the score is to `0.0`, the more likely that face encoding is the same as those described in the list of known encodings.

```
known_encodings = [...]
score = encoder.score_encoding(face_encoding, known_encodings)
```

Note that `known_encodings` takes a list of encodings rather than a single encoding. This list of encodings should always be encodings of the same person. If you start passing in various encodings from different people, the produced score won't make any sense.

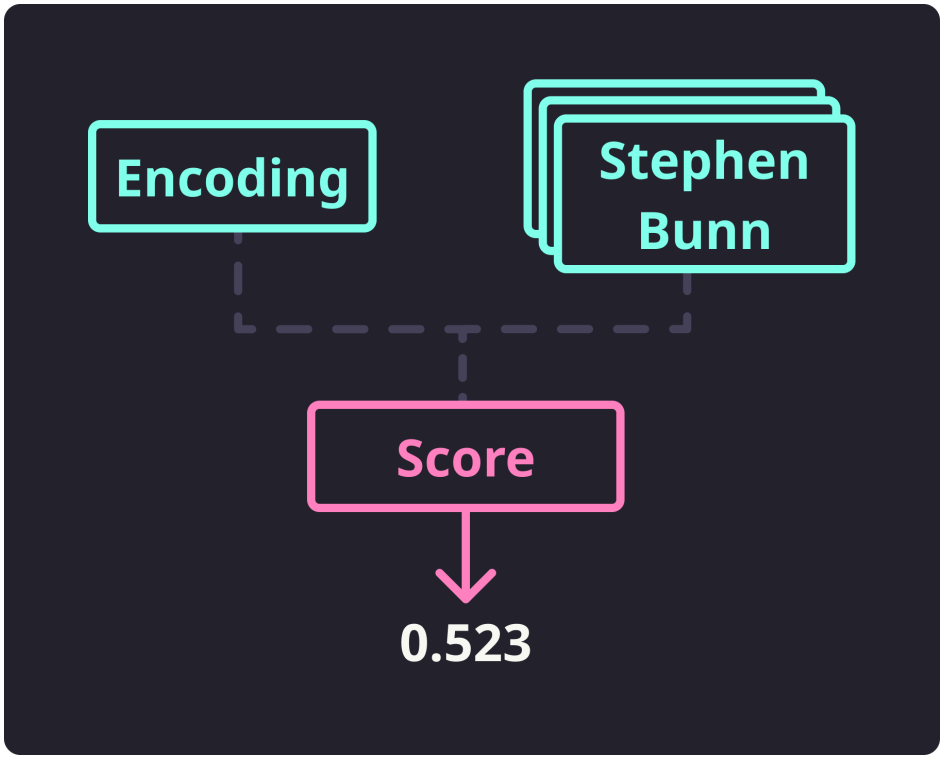

Fig. 16: Score Flow

It's probably easier to show what some very basic face recognition looks like. Below is an example of some stored face encodings, a few of myself and a few of Terry Crews. The following script will iterate over the stored face encodings and determine the best fit for each detected face my webcam stream.

```
from facelift.capture import iter_stream_frames
from facelift.window import opencv_window
from facelift.draw import draw_text
from facelift.detect import BasicFaceDetector
from facelift.encode import BasicFaceEncoder

detector = BasicFaceDetector()
encoder = BasicFaceEncoder()

```

(continues on next page)

(continued from previous page)

```
# known encodings for specific faces
# trimmed out the actual encodings to preserve some readability
face_storage = {
   "Stephen Bunn": [...],
   "Terry Crews": [...]
}

with opencv_window() as window:
    for frame in iter_stream_frames():
        for face in detector.iter_faces(frame):
            face_encoding = encoder.get_encoding(frame, face)

            # collect scores for our storage of known encodings
            # this could be further optimized using multi-threading or better
            # storage mechanisms
            scores = []
            for name, known_encodings in face_storage.items():
                score = encoder.score_encoding(face_encoding, known_encodings)
                scores.append((score, name))

            # printing out score results for our known faces so you can see
            # what kind of scores are being produced
            print(
                "\r" + ", ".join([
                    f"{name} ({score:1.2f})" for score, name in scores
                ]),
                end="",
            )

            # get the best scored name for this face
            best_name = min(scores, key=lambda x: x[0])[-1]

            # draw the best name right above the face
            frame = draw_text(
                frame,
                best_name,
                *face.rectangle,
                font_scale=0.5,
                color=(255, 0, 0),
                x_position=Position.CENTER,
                y_position=Position.START,
                allow_overflow=True,
            )

    window.render(frame)
```

You can see that when we are printing results in the terminal, the score for my name is actually further from `0.0` than the score we get for Terry. This is because I'm actually subtracting from `1.0` in this recording which is something I forgot to remove and I'm too lazy to remake the recoding. You can ignore the numbers being written to `stdout` in this case as they contradict what you should be expecting from *`score_encoding()`*.

And here is a run of the same script but with both me and a picture of Terry. You can see that my face tone is darker as I had to close some blinds to avoid screen glare off of my phone. That screen glare was causing some obvious issues with detecting Terry's face.

Of course, you can optimize this a bit, but for the purposes of demonstration we left it as simple and readable as possible.

At this point you should have enough details to get started using some of the features available in this package. If you find anything that you think could be improved, you can interact with development in the Facelift GitHub repository.

Thanks for reading through these docs!

CHAPTER

THREE

CONTRIBUTING

Important: When contributing to this repository, please adhere to our *Code of Conduct* and first discuss the change you wish to make via an issue **before** submitting a pull request.

3.1 Local Development

The following sections will guide you through setting up a local development environment for working on this project package. **At the very least**, make sure that you have the necessary pre-commit hooks installed to make sure that all commits are pristine before they make it into the change history.

3.1.1 Installing Python

Note: If you already have Python 3.7+ installed on your local system, you can skip this step completely.

Installing Python should be done through pyenv. To first install `pyenv` please follow the guide they provided at https://github.com/pyenv/pyenv#installation. When you finally have `pyenv` you should be good to continue on.

```
$ pyenv --version
pyenv x.x.x
```

Now that you have `pyenv` we can install the necessary Python version. This project's package depends on Python 3.7+, so we can request that through `pyenv`.

```
$ pyenv install 3.7  # to install Python 3.7+
...

$ pwd
/PATH/TO/CLONED/REPOSITORY/project-name
$ pyenv local 3.7  # to mark the project directory as needing Python 3.7+
...

$ pyenv global 3.7  # if you wish Python 3.7 to be aliased to `python` everywhere
...
```

After installing and marking the repository as requiring Python 3.7+ you should be good to continue on installing the project's dependencies.

3.1.2 Virtual Environment

We use Poetry to manage both our dependencies and virtual environments. Setting up `poetry` just involves installing it through `pip` as a user-level dependency.

```
$ pip install --user poetry
Collecting poetry
Downloading poetry-x.x.x-py2.py3-non-any.whl
...
```

You can quickly setup your entire development environment by running the installation process from poetry.

```
$ poetry install
Installing dependencies from lock file
...
```

This with create a virtual environment for you and install the necesary development dependencies. From there you can jump into a subshell using the newly created virtual environment using the `shell` subcommand.

```
$ poetry shell
spawning shell within ~/.local/share/virtualenvs/my-project-py3.7
...

$ exit # when you wish to exit the subshell
```

From this shell you have access to all the necessary development dependencies installed in the virutal environment and can start actually writing and running code within the client package.

3.1.3 Style Enforcement

This project's preferred styles are fully enforced through pre-commit hooks. In order to take advantage of these hooks please make sure that you have `pre-commit` and the configured hooks installed in your local environment.

Installing `pre-commit` is done through `pip` and should be installed as a user-level dependency as it adds some console scripts that all projects using `pre-commit` will need.

```
$ pip install --user pre-commit
Collecting pre-commit
Downloading pre_commit-x.x.x-py2.py3-none-any.whl
...

$ pre-commit --version
pre-commit 2.4.0
```

Once `pre-commit` is installed you should also install the hooks into the cloned repository.

```
$ pwd
/PATH/TO/CLONED/REPOSITORY/project-name

$ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

After this you should be good to continue on. These installed hooks will do a first-time setup when you attempt your next commit to build hook environments. Changes that violate the defined style specifications in `setup.cfg` and `pyproject.toml` will cause the commit to fail and will likely make the necessary changes to added / changed files to be written to the failing files.

This will give you the opprotunity to view the changes the hooks made to the failing files and add the new changes to the commit in order to make the commit pass. It also gives you the opprotunity to make tweaks to the autogenerated changes to make them more human accessible (only if necessary).

3.1.4 Editor Configuration

We also have some specific settings for editor configuration via editorconfig. We recommend you install the appropriate plugin for your editor of choice if your editor doesn't already natively support `.editorconfig` configuration files.

3.1.5 Project Tasking

All of our tasks are built and run through invoke which is basically just a more advanced (a little too advanced) Python alternative to make. The only reason we are using this utility is because I know how it works and I already had most of the necessary tasks defined from other projects.

From within the Poetry subshell, you can access and run these commands through the provided `invoke` development dependency.

```
$ invoke --list
Available tasks:

   build                  Build the project.
   clean                  Clean the project.
   lint                   Lint the project.
   profile                Run and profile a given Python script.
   test                   Test the project.
   docs.build             Build docs.
   docs.build-news        Build towncrier newsfragments.
   docs.clean             Clean built docs.
   docs.view              Build and view docs.
   linter.black           Run Black tool check against source.
   linter.flake8          Run Flake8 tool against source.
   linter.isort           Run ISort tool check against source.
   linter.mypy            Run MyPy tool check against source.
   package.build          Build pacakge source files.
   package.check          Check built package is valid.
   package.clean          Clean previously built package artifacts.
   package.coverage       Build coverage report for test run.
   package.format         Auto format package source files.
   package.requirements   Generate requirements.txt from Poetry's lock.
   package.stub           Generate typing stubs for the package.
   package.test           Run package tests.
   package.typecheck      Run type checking with generated package stubs.
```

You can run these tasks to do many miscellaneous project tasks such as building documentation.

```
$ invoke docs.build
[docs.build] ... building 'html' documentation
Running Sphinx v3.0.3
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
```

(continues on next page)

(continued from previous page)

```
no targets are out of date.
build succeeded.

The HTML pages are in build/html.
```

All of these tasks should just work right out of the box, but something might break eventually after required tooling gets enough major updates.

3.2 Opening Issues

Issues should follow the included `ISSUE_TEMPLATE` found in `.github/ISSUE_TEMPLATE.md`.

- **Issues should contain the following sections:**
  - Expected Behavior
  - Current Behavior
  - Possible Solution
  - Steps to Reproduce (for bugs)
  - Context
  - Your Environment

These sections help the developers greatly by providing a large understanding of the context of the bug or requested feature without having to launch a full fleged discussion inside of the issue.

3.3 Creating Pull Requests

Pull requests should follow the included `PULL_REQUEST_TEMPLATE` found in `.github/PULL_REQUEST_TEMPLATE.md`.

- Pull requests should always be from a `topic` / `feature` / `bugfix` (left side) branch.
  Pull requests from master branches will not be merged.
- Pull requests should not fail our requested style guidelines or linting checks.

CHAPTER

FOUR

CODE OF CONDUCT

4.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

4.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

4.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

4.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at stephen@bunn.io. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

4.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

4.6.1 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

4.6.2 2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

4.6.3 3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4.6.4 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

4.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

CHAPTER

FIVE

CHANGELOG

All notable changes to this project will be documented in this file.
The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

5.1 0.2.1 (*2020-10-30*)

5.1.1 Bug Fixes

- Fixing the release task and some inconsistencies that were causing the *download_data()* function to raise a ValueError.

5.2 0.2.0 (*2020-10-30*)

5.2.1 Miscellaneous

- Due to PyPi's upload limit of ~100MB, we cannot bundle pre-trained models along with the built package. We are now building a process for acquiring these models around an included function that will attempt to fetch the latest released models from GitHub releases.

  The method *download_data()* should be a quick initial setup task when attempting to use this module. This task will download the necessary data to use the included detectors and encoders.

  More details about what is necessary for this release process is and should (in the future) be documented in the *facelift._data* module.

5.3 0.1.0 (*2020-10-27*)

5.3.1 Miscellaneous

- The initial release doesn't have a super detailed list of introduced features or bugfixes as this project was pulled together from other side projects I've had in the past. Below I'll list the important features that we are starting out with. Future additions should result in a history of news fragments that get aggregated into this changelog.

  Starting features:

  1. **Face feature detection with a few bundled models.**

  - Basic face feature detection (eyes and nose)
  - Partial face feature detection (trained model produced by `dlib`)
  - Full face feature detection (third party trained model)

2. **Face recognition with a bundled ResNet produced by `dlib` to produce face encoding.**
  - Includes basic Euclidean distance scoring to find similar faces.
3. **Wrappers for OpenCV frame capturing.**
  - Generators for frames from written media files.
  - Generators for frames from streaming devices (webcams).
4. **Wrappers for OpenCV windows.**
  - Context managers for named window management.
5. **Wrappers for OpenCV common frame transformations**
  - Scaling, resizing, rotating, cutting, copying, etc…
6. **Wrappers for OpenCV canvas drawing features**
  - Helper functions for drawing points, lines, polygons, text, etc…
7. **Example *`helpers`* module for basic face normalization.**
  - Gives a basic re-implementation of `dlib`'s `get_face_chip()` method.

CHAPTER

SIX

LICENSE

```
ISC License

Copyright (c) 2020, Stephen Bunn <stephen@bunn.io>

Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

CHAPTER

SEVEN

ATTRIBUTION

A lot of the logic that makes up this package is pulled from other projects or is using resources that other developers have produced. Below are callouts to the projects or resources that were used to create this package.

Facelift Icon

— Created by **Delwar Hossain** pulled from Noun Project

81 Point Face Feature Model

— Created by **codeniko** pulled from codeniko/shape_predictor_81_face_landmarks

This model is used as part of the `FullFaceDetector` to allow for detecting the `FOREHEAD` feature. Thanks to codeniko for traning this model.

Image Transformation Utilities

— Created by **josebr1** pulled from josebr1/imutils

A load of utilities and image transformations were inspired by and sometimes directly pulled the work done in this project. Thanks to all the contributors who threw these together. It made refactoring them to fit for my own design and use-case much easier.

Facial Recognition

— Created by **ageitgey** pulled from ageitgey/face_recognition

A fair chunk of the logic used for facial recognition was inspired from the work done in this project. Thanks to all the contributors who worked on building this iteration of a facial recognition package.

CHAPTER

EIGHT

PROJECT REFERENCE

8.1 Facelift Package

8.1.1 facelift.types

Contains module-wide used types.

facelift.types.**Frame**
An aliased type for a basic numpy array that gets given to use via OpenCV.

Type NDArray[(Any, Any, 3), UInt8]

facelift.types.**Point**
A single x, y coordinate that describes a single positional point.

Type NDArray[(2,), Int32]

facelift.types.**PointSequence**
A sequence of points that is typically used to describe a face feature or a line during rendering.

Type NDArray[(Any, 2), Int32]

facelift.types.**Encoding**
A 128 dimension encoding of a detected face for a given frame.

Type NDArray[(128,), Int32]

facelift.types.**Detector**
Callable that takes a frame and an upsample count and discovers the bounds of a face within the frame.

Type Callable[[*Frame*, int], *PointSequence*]

facelift.types.**Predictor**
Callable which takes a frame and detected face bounds to discover the shape and features within the face.

Type Callable[[*Frame*, dlib.rectangle], dlib.full_object_detection]

class facelift.types.**Encoder**(**args*, ***kwds*)
Protocol class for dlib.face_recognition_model_v1..

compute_face_descriptor(*frame*, *face*, *num_jitters=0*, *padding=0.25*)
Compute a descriptor for a detected face frame.

Parameters

- **frame** (*Frame*) – The frame containing just the detected face.
- **face** (dlib.full_object_detection) – The raw detected face bounds within the given face frame.

- **num_jitters** (*int*) – The number of jitters to run through the dector projection. Defaults to 0.
- **padding** (*float*) – The default padding around the face. Defaults to 0.25.

Returns The face descriptor (encoding).

Return type dlib.vector

class facelift.types.**Face**(*raw*, *landmarks*, *frame*)

Describes a detected face.

Parameters

- **raw** (dlib.full_object_detection) – The raw dlib object detection container.
- **landmarks** (Dict[*FaceFeature*, *PointSequence*]) – Mapping of extracted face features to the sequence of points describing those features.
- **frame** (*Frame*) – The base non-normalized cropped frame of just the face.

property rectangle

Point sequence representation of the detected face's bounds.

This property is useful for properly positioning text around the detected face as the *draw_text()* needs a text container to be defined.

Returns A sequence of 2 points indicating the top-left and bottom-right corners of the detected face's bounds.

Return type *PointSequence*

class facelift.types.**FaceFeature**(*value*)

Enumeration of features of a face that we can detect.

NOSE

The nose of a face.

JAW

The jaw line of a face.

MOUTH

The external bounds of the mouth of a face.

INNER_MOUTH

The internal bounds of the mouth of a face.

RIGHT_EYE

The external and internal bounds of the right eye of a face.

LEFT_EYE

The external and internal bounds of the left eye of a face.

RIGHT_EYEBROW

The right eyebrow of a face.

LEFT_EYEBROW

The left eyebrow of a face.

FOREHEAD

The forehead curvature of a face.

class facelift.types.**MediaType**(*value*)

Enumeration of acceptable media types for processing.

IMAGE
Defines media that contains just a single frame to process.

VIDEO
Defines media that contains a known number of frames to process.

STREAM
Defines media that contains an unknown number of frames to process.

8.1.2 facelift.capture

Contains helpers and managers for capturing content from various sources.

Among the included functions, *iter_media_frames()* and *iter_stream_frames()* should really be all you ever care about. With these two functions you can iterate over either some image or video (as supported by OpenCV) or frames streamed directly from a webcam. The frames output by these generators are `numpy` arrays that are considered *Frame* instances and are used throughout the project.

For example, if I had a video file `~/my-file.mp4` and wanted to iterate over all available frames within the video, I would do use *iter_media_frames()* like the following:

```
from pathlib import Path
from facelift.capture import iter_media_frames

MY_FILE = Path("~/my-file.mp4").expanduser()
for frame in iter_media_frames(MY_FILE):
    print(frame)
```

The same works for images, however only 1 frame will ever be yielded from the generator.

If you want to instead iterate over the frames from a webcam, you should use the *iter_stream_frames()* like the following:

```
from facelift.capture import iter_stream_frames
# will default the device id to a value of 0
# this means OpenCV will attempt to discover the first available webcam
for frame in iter_stream_frames():
    print(frame)

# if you have 2 webcams enabled and want to instead use the 2nd one, you should
# specify a device index of 1 like this
for frame in iter_stream_frames(1):
    print(frame)
```

facelift.capture.**file_capture**(*filepath*)
Context manager to open a given filepath for frame capture.

This is just a simple context manager wrapper around the base *media_capture()* manager to ensure that a given `filepath` exists and is a supported media type before attempting to build a capture around it.

Examples

```
>>> from pathlib import Path
>>> from facelift.capture import file_capture
>>> MY_FILEPATH = Path("~/my-file.mp4").expanduser()
>>> with file_capture(MY_FILEPATH) as capture:
...     print(capture)
<VideoCapture 0x1234567890>
```

Parameters **filepath** (*Path*) – The filepath to open for capture

Raises

- **FileNotFoundError** – When the given filepath doesn't exist
- **ValueError** – When the given filepath is not a supported media type

Yields *cv2.VideoCapture* – A capturer that allows for reading frames from the given media filepath

Return type Generator[VideoCapture, None, None]

facelift.capture.**iter_media_frames**(*media_filepath*, *loop=False*)

Iterate over frames from a given supported media file.

Examples

```
>>> from pathlib import Path
>>> from facelift.capture import iter_media_frames
>>> MEDIA_PATH = Path("~/my-media.mp4").expanduser()
>>> for frame in iter_media_frames(MEDIA_PATH):
...     # do something with the frame
```

Parameters

- **media_filepath** (*Path*) – The filepath to the media to read frames from.
- **loop** (*bool*) – Flag that indicates if capture should reset to starting frame once all frames have been read. Defaults to False

Yields *Frame* – A frame read from the given media file

Return type Generator[Type[ndarray], None, None]

facelift.capture.**iter_stream_frames**(*stream_type=None*)

Iterate over frames from a given streaming device.

By default this iterator will attempt to connect to the first available webcam and yield the webcam's streamed frames. You can specify the appropriate device index 0-99 (0 being the default), or a custom stream type defined by the OpenCV video IO enum.

Examples

```
>>> from facelift.capture import iter_stream_frames
>>> # iterate over frames available from the second available webcam
>>> for frame in iter_stream_frames(1):
...     # do something with the frame
```

Parameters **stream_type** (*Optional[int], optional*) – The stream type to attempt to open.

Yields *Frame* – A read frame from the given streaming device

Return type Generator[Type[ndarray], None, None]

facelift.capture.**media_capture**(*media*, *media_type*)

General purpose media capture context manager.

This context manager is basically just a wrapper around the provided VideoCapture constructor along with some capturing destruction logic. The provided media can either be a filepath to capture frames off of or a device id as defined by the OpenCV video IO enum.

In most all cases where you just want to build a capture off of your default webcam, you should just be giving a media of 0.

Examples

```
>>> # build a media capture for a specific media file
>>> from facelift.capture import media_capture
>>> with media_capture("/home/a-user/Desktop/test.mp4") as capture:
...     print(capture)
<VideoCapture 0x1234567890>
```

```
>>> # build a media capture around the first available webcam
>>> from facelift.capture import media_capture
>>> with media_capture(0) as capture:
...     print(capture)
<VideoCapture 0x1234567890>
```

Parameters

- **media** (*Union[str, int]*) – The media to build a capturer for
- **media_type** (*MediaType*) –

Raises **ValueError** – On failure to open the given media for capture

Yields *cv2.VideoCapture* – A capturer that allows for reading sequential frames

Return type Generator[VideoCapture, None, None]

facelift.capture.**stream_capture**(*stream_type=None*)

Context manager to open a stream for frame capture.

By default this context manager will just attempt to connect to open capturing on any available webcams or connected cameras. You can get more specific about what device you would like to open a capturer on by supplying a different stream type. These stream types come directly from the OpenCV video IO enum.

Examples

```
>>> # build a frame capture from the first available webcam
>>> from facelift.capture import stream_capture
>>> with stream_capture() as capture:
...     print(capture)
<VideoCapture 0x1234567890>
```

```
>>> # build a frame capture from the second available webcam
>>> from facelift.capture import stream_capture
>>> with stream_capture(1) as capture:
...     print(capture)
<VideoCapture 0x1234567890>
```

Parameters **stream_type** (*Optional[int], optional*) – The stream type to open

Raises **ValueError** – When the given stream device fails to be opened for capture

Yields *cv2.VideoCapture* – A capturer that allows for reading frames from the defined stream type

Return type Generator[VideoCapture, None, None]

8.1.3 facelift.transform

Contains some common necessary frame transformation helper methods.

These transformation methods are useful for optimizing face detection in frames. Typically face detection takes much longer the more pixels there are to consider. Therefore, using *scale()* or *resize()* will help you speed up detection.

These helper transforms can be composed together to produce apply multiple operations on a single frame. For example, if we wanted to first downscale by half and then rotate a frame by 90 degrees, we could do something like the following:

```
from facelift.transform import rotate, scale
transformed_frame = rotate(scale(frame, 0.5), 90)
```

facelift.transform.**DEFAULT_INTERPOLATION**
The default type of interpolation to use in transforms that require an interpolation method. Defaults to cv2.INTER_AREA.

Type int

facelift.transform.**adjust**(*frame*, *brightness=None*, *sharpness=None*)
Adjust the brightness or sharpness of a frame.

Examples

```
>>> from facelift.transform import adjust
>>> sharper_frame = adjust(frame, sharpness=1.4)
>>> brighter_frame = adjust(frame, brightness=10)
>>> sharper_and_brighter_frame = adjust(frame, sharpness=1.4, brightness=10)
```

Parameters

- **frame** (*Frame*) – The frame to adjust
- **brightness** (*Optional[int], optional*) – The new brightness of the frame (can be negative, default is 0). Defaults to 0.
- **sharpness** (*Optional[float], optional*) – The new sharpness of the frame (0.0 is black, default is 1.0). Defaults to 1.0.

Returns The newly adjusted frame

Return type *Frame*

facelift.transform.**copy**(*frame*)

Copy the given frame to a new location in memory.

Examples

```
>>> from facelift.transform import copy
>>> copied_frame = copy(frame)
>>> assert frame == copied_frame
>>> assert frame is not copied_frame
```

Parameters **frame** (*Frame*) – The frame to copy

Returns An exact copy of the given frame

Return type *Frame*

facelift.transform.**crop**(*frame*, *start*, *end*)

Crop the given frame between two top-left to bottom-right points.

Examples

Crop a frame from the first pixel to the center pixel.

```
>>> from facelift.transform import crop
>>> assert frame.shape[:1] == [512, 512]
>>> cropped_frame = crop(frame, (0, 0), (256, 256))
>>> assert cropped_frame.shape[:1] == [256, 256]
```

Parameters

- **frame** (*Frame*) – The frame to crop
- **start** (*Tuple[int, int]*) – The top-left point to start the crop at
- **end** (*Tuple[int, int]*) – The bottom-right point to end the crop at

Raises `ValueError` – When the given starting crop point appears after the given ending crop point

Returns The newly cropped frame

Return type *Frame*

facelift.transform.**flip**(*frame*, *x_axis=False*, *y_axis=False*)

Flip the given frame over either or both the x and y axis.

Examples

```
>>> from facelift.transform import flip
>>> vertically_flipped_frame = flip(frame, x_axis=True)
>>> horizontally_flipped_frame = flip(frame, y_axis=True)
>>> inverted_frame = flip(frame, x_axis=True, y_axis=True)
```

Parameters

- **frame** (*Frame*) – The frame to flip
- **x_axis** (*bool, optional*) – Flag indicating the frame should be flipped vertically. Defaults to False.
- **y_axis** (*bool, optional*) – Flag indicating the frame should be flipped horizontally. Defaults to False.

Returns The newly flipped frame

Return type *Frame*

facelift.transform.**grayscale**(*frame*)

Convert the given frame to grayscale.

This helper is useful *sometimes* for classification as color doesn't matter as much during face encoding.

Examples

```
>>> from facelift.transform import grayscale
>>> grayscale_frame = grayscale(bgr_frame)
```

Parameters **frame** (*Frame*) – The BGR frame to convert to grayscale

Returns The newly grayscaled frame

Return type *Frame*

facelift.transform.**resize**(*frame*, *width=None*, *height=None*, *lock_aspect=True*, *interpolation=cv2.INTER_AREA*)

Resize a given frame to a given width and/or height.

- If both width and height are given, the frame will be resized accordingly.
- If only one of width or height is given, the frame will be resized according to the provided dimension (either width or height).
  - As long as `lock_aspect` is truthy, the unprovided dimension will be adjusted to maintain the original aspect-ratio of the frame.

– If `lock_aspect` is falsy, the resize operation will only scale the provided dimension while keeping the original size of the unprovided dimension.

Examples

Resize a frame's width while keeping the height relative:

```
>>> from facelift.transform import resize
>>> assert frame.shape[:1] == [512, 512]
>>> resized_frame = resize(frame, width=256, lock_aspect=True)
>>> assert resized_frame.shape[:1] == [256, 256]
```

Resize a frame's width while keeping the original height:

```
>>> from facelift.transform import resize
>>> assert frame.shape[:1] == [512, 512]
>>> resized_frame = resize(frame, width=256, lock_aspect=False)
>>> assert resized_frame.shape[:1] == [512, 256]
```

Resize both a frame's width and height:

```
>>> from facelift.transform import resize
>>> assert frame.shape[:1] == [512, 512]
>>> resized_frame = resize(frame, width=256, height=128)
>>> assert resized_frame.shape[:1] == [128, 256]
```

Parameters

- **frame** (*Frame*) – The frame to resize
- **width** (*Optional[int], optional*) – The exact width to resize the frame to.
- **height** (*Optional[int], optional*) – The exact height to resize the frame to.
- **lock_aspect** (*bool, optional*) – Whether to keep the width and height relative when only given one value. Defaults to True.
- **interpolation** (*int, optional*) – The type of interpolation to use in the resize operation. Defaults to *DEFAULT_INTERPOLATION*.

Returns The newly resized frame

Return type *Frame*

`facelift.transform.rgb`(*frame*)

Convert the given frame to RGB.

This helper transform is typically needed when working with other image processing libraries such as pillow as they work in RGB coordinates while OpenCV works in BGR coordinates.

Examples

```
>>> from facelift.transform import rgb
>>> rgb_frame = rgb(bgr_frame)
```

Parameters **frame** (*Frame*) – The BGR frame to convert to RGB

Returns The new RGB frame

Return type *Frame*

facelift.transform.**rotate**(*frame*, *degrees*, *interpolation=cv2.INTER_AREA*)

Rotate a frame while keeping the whole frame visible.

Examples

```
>>> from facelift.transform import rotate
>>> rotated_90 = rotate(frame, 90)
>>> rotated_neg_90 = rotate(frame, -90)
```

Warning: This transform typically will produce larger frames since we are producing a rotated frame while keeping the original frame completely visible. This means if we do a perfect 45 degree rotation on a 512x512 frame we will produce a 724x724 frame since the 512x512 frame is now on a angle that requires a larger container.

Be cautious when using rotation. Most of the time you do not need to rotate on any angles other than 90, 180, and 270 for decent face detection. However, this isn't *always* true.

Parameters

- **frame** (*Frame*) – The frame to rotate
- **degrees** (*int*) – The number of degrees to rotate the given frame
- **interpolation** (*int, optional*) – The type of interpolation to use in the produced rotation matrix. Defaults to *DEFAULT_INTERPOLATION*.

Returns The newly rotated frame

Return type *Frame*

facelift.transform.**scale**(*frame*, *factor*, *interpolation=cv2.INTER_AREA*)

Scale a given frame down or up depending on the given scale factor.

Examples

Downscaling a frame can be performed with a `scale` factor >0 and <1. For example, scaling a frame to half of its original size would require a scale factor of 0.5.

```
>>> from facelift.transform import scale
>>> assert frame.shape[:1] == [512, 512]
>>> downscaled_frame = scale(frame, 0.5)
>>> assert downscaled_frame.shape[:1] == [256, 256]
```

Upscaling a frame with this method is **very** naive and suboptimal. However, any value >1 will result in a upscaled frame. For example, scaling a frame to double its original size would require a scale factor of 2.

```
>>> from facelift.transform import scale
>>> assert frame.shape[:1] == [512, 512]
>>> upscaled_frame = scale(frame, 2)
>>> assert upscaled_frame.shape[:1] == [1024, 1024]
```

Following this logic, a scale factor of 1 would result in absolutely no change to the given frame.

Warning: This transformation will return the **exact same frame instance** as the one provided through the `frame` parameter in the following cases:

1. If a factor of exactly `1` is given. In this case the scale operation would result in no change.
2. The given frame has factor less than `1` a width or height of 1px. In this case we are attempting to scale down the given frame and we cannot scale down the frame any further without producing a 0px frame.

Parameters

- **frame** (*Frame*) – The frame to scale
- **factor** (*float*) – The factor to scale the given frame
- **interpolation** (*Optional[int], optional*) – The type of interpolation to use in the scale operation. Defaults to *DEFAULT_INTERPOLATION*.

Raises **ValueError** – When the given scale factor is not positive

Returns The newly scaled frame

Return type *Frame*

facelift.transform.**translate**(*frame*, *delta_x=None*, *delta_y=None*, *interpolation=cv2.INTER_AREA*)

Translate the given frame a specific distance away from its origin.

Examples

```
>>> from facelift.transform import translate
>>> translated_neg_90_x_frame = translate(frame, delta_x=-90)
```

Important: This translation retains the original size of the given frame. So a 512x512 frame translated 90px on the x-axis will still be 512x512 and space where the frame use to take up will be essentially nulled out.

Parameters

- **frame** (*Frame*) – The frame to translate
- **delta_x** (*Optional[int], optional*) – The pixel distance to translate the frame on the x-axis.
- **delta_y** (*Optional[int], optional*) – The pixel distance to translate the frame on the y-axis.

- **interpolation** (*int, optional*) – The type of interpolation to use during the translation. Defaults to *DEFAULT_INTERPOLATION*.

Returns The newly translated frame

Return type *Frame*

8.1.4 facelift.magic

Contains helpers and enums used to guess the type of media that is being processed.

This module utilizes python-magic which in turn uses libmagic to guess the appropriate mimetype of some byte buffer.

facelift.magic.**DEFAULT_MAGIC_BUFFER_SIZE**
The default number of bytes to try and read from when making a guess at the mimetype of some file.

Type int

facelift.magic.**get_media_type**(*media_filepath*, *buffer_size=None*, *validate=False*)
Try and determine the media type for content at the given filepath.

Parameters

- **media_filepath** (*Path*) – The filepath to guess the media type of
- **buffer_size** (*Optional[int], optional*) – The number of bytes to use for guessing the media type of the given file. Defaults to the value of *DEFAULT_MAGIC_BUFFER_SIZE*.
- **validate** (*bool, optional*) – If truthy, a `ValueError` will be raised if the given file's mimetype does not match a supported *MediaType*. Defaults to False.

Raises

- **FileNotFoundError** – When the provided filepath does not exist
- **ValueError** – When `validate` is truthy and the given filepath does not match a supported *MediaType*

Returns The appropriate media type enum attribute for the given filepath, if a successful guess and media type match is made

Return type Optional[MediaType]

facelift.magic.**get_mimetype**(*media_filepath*, *buffer_size=None*)
Try and determine the mimetype for content at the given filepath.

Parameters

- **media_filepath** (*Path*) – The filepath to guess the mimetype of
- **buffer_size** (*Optional[int], optional*) – The number of bytes to use for guessing the mimetype of the given file. Defaults to the value of *DEFAULT_MAGIC_BUFFER_SIZE*.

Raises **FileNotFoundError** – When the provided filepath does not exist

Returns The guessed mimetype if a guess can be safely made

Return type Optional[str]

8.1.5 facelift.detect

Contains the available bulitin face detectors.

The included detectors will handle the necessary process for taking a read frame and discovering all the available faces and included landmarks. If you have a custom `face_landmarks` model, you can inherit from *BaseLandmarkDetector* to detect and return faces using a custom model.

Examples

```
>>> from facelift.detect import BasicFaceDetector
>>> from facelift.capture import iter_media_frames
>>> detector = BasicFaceDetector()
>>> for frame in iter_media_frames(MEDIA_FILEPATH):
...     for face in detector.iter_faces(frame):
...         print(face)
```

class facelift.detect.**BaseLandmarkDetector**

An abstract landmark detector class that each landmark model should inherit from.

Raises

- **NotImplementedError** – If the `model_filepath` property is not implemented
- **NotImplementedError** – If the `landmark_slices` property is not implemented

detector

Detector to use in face bounds detection.

Returns The detector callable.

Return type *Detector*

get_landmarks(*points*)

Get the mapping of face features and point sequences for extracted points.

Parameters **points** (*PointSequence*) – The sequence of extracted points from dlib.

Returns The dictionary of face features and point sequences.

Return type Dict[*FaceFeature*, *PointSequence*]

iter_faces(*frame*, *upsample=0*)

Iterate over detected faces within a given *Frame*.

Examples

Get detected faces from the first available webcam.

```
>>> from facelift.capture import iter_stream_frames
>>> from facelift.detect import BasicFaceDetector
>>> detector = BasicFaceDetector()
>>> for frame in iter_stream_frames():
...     for face in detector.iter_faces(frame):
...         print(face)
```

Parameters

- **frame** (*Frame*) – The frame to detect faces in.

- **upsample** (*int, optional*) – The number of times to scale up the image before detecting faces. Defaults to 0.

Yields *Face* – A detected face within the image, this has no guarantee of order if multiple faces are detected

Return type Generator[*Face*, None, None]

abstract property landmark_slices
Property mapping of facial features to face point slices.

Raises **NotImplementedError** – Must be implemented by subclasses

Return type Dict[*FaceFeature*, Tuple[int, int]]

abstract property model_filepath
Property filepath to the landmarks model that should be used for detection.

Raises **NotImplementedError** – Must be implemented by subclasses

Return type Path

predictor
Predictor to use in face landmark detection.

Returns The predictor callable.

Return type *Predictor*

static shape_to_points(*shape*, *dtype='int'*)
Convert dlib shapes to point sequences.

Example

After getting a detected face shape from dlib, we need to convert it back into a `numpy.ndarray` so OpenCV can use it.

```
>>> from facelift.detect import BasicFaceDetector
>>> detector = BasicFaceDetector()
>>> for face_bounds in detector.detector(frame, 0):
...     face_shape = detector.predictor(frame, face_bounds)
...     face_features = detector.shape_to_points(face_shape)
```

Parameters

- **shape** (dlib.full_object_detection) – The detected dlib shape.
- **dtype** (*str, optional*) – The point type to use when converting the given shape to points. Defaults to "int".

Returns The newly created sequence of points.

Return type *PointSequence*

static slices_to_landmarks(*points*, *feature_slices*)
Group point sequences to features based on point index slices.

Helper function to automatically group features when given the feature slice definition. This feature slice definition is a basic way to easily categorize the features discovered from the dlib predictor as an actual *FaceFeature*.

Examples

```
>>> from facelift.detect import BasicFaceDetector
>>> detector = BasicFaceDetector()
>>> for face_bounds in detector.detector(frame, 0):
...     face_shape = detector.predictor(frame, face_bounds)
...     face_features = detector.shape_to_points(face_shape)
...     grouped_features = detector.slices_to_landmarks(face_features)
```

Parameters

- **points** (*PointSequence*) – The points to extract feature sequences from.
- **feature_slices** (Dict[*FaceFeature*, Tuple[int, int]]) – A dictionary of *FaceFeature* and slice tuples.

Returns The dictionary of features and grouped point sequences.

Return type Dict[*FaceFeature*, *PointSequence*]

class facelift.detect.**BasicFaceDetector**

Basic face detector.

This face detector gives single point positions for the outside of both eyes and the philtrum (right beneath the nose). For rendering, these facial features must be rendered as points rather than lines.

This model is useful for just finding faces and getting normalized face frames. This model is not useful for emotion, perspective, or face state detection.

class facelift.detect.**FullFaceDetector**

Full face detector.

This face detector detects all available frontal face features. This model can be used for most anything, but may not be the most efficient. If you are just trying to detect faces, you should probably use the *BasicFaceDetector* instead.

get_landmarks(*points*)

Get the mapping of face features and point sequences for extracted points.

Parameters **points** (*PointSequence*) – The sequence of extracted points from dlib.

Returns The dictionary of face features and point sequences.

Return type Dict[*FaceFeature*, *PointSequence*]

class facelift.detect.**PartialFaceDetector**

Partial face detector.

This face detector detects all features of a face except for the forehead. This model is useful for most any purpose. However, if all you are doing is detecting faces, you should probably use the *BasicFaceDetector* instead.

facelift.detect.**get_detector**()

Build the generic detector callable.

This detector comes directly from the dlib FHOG frontal face detector.

Returns The new callable to detect face bounds

Return type *Detector*

facelift.detect.**get_predictor**(*model_filepath*)

Build a predictor callable for a given landmark model.

Parameters **model_filepath** (*Path*) – The path to the landmark model

Raises **FileNotFoundError** – If the given model filepath does not exist

Returns The new callable to predict face shapes

Return type *Predictor*

8.1.6 facelift.encode

Contains the available builtin face encoders.

The included encoders will handle the necessary steps to take a given frame and detected face to generate an encoding that can be used for future recognition. I highly recommend that you use the *BasicFaceDetector* if attempting to encode faces as it is lightweight and other detectors don't provide any added benefit to face recognition.

Examples

```
>>> from facelift.capture import iter_media_frames
>>> from facelift.detect import BasicFaceDetector
>>> from facelift.encode import BasicFaceEncoder
>>> detector = BasicFaceDetector()
>>> encoder = BasicFaceEncoder()
>>> for frame in iter_media_frames(MEDIA_FILEPATH):
...     for face in detector.iter_faces(frame):
...         face_encoding = encoder.get_encoding(frame, face)
```

Important: Faces detected from the *FullFaceDetector* cannot be encoded as the model this detector uses is trained by a third party and not able to be processed by `dlib`'s default ResNet model. Please only use faces detected using the *BasicFaceDetector* or the *PartialFaceDetector* for building face encodings.

I would **highly** recommend that you use the *BasicFaceDetector* in all cases where you are performing encoding. The trained detection model for this basic detector is ~5MB whereas the alternative is >90MB. Using a heavier model will cause slowdown when simply trying to recognize multiple faces in a single frame.

facelift.encode.**DEFAULT_ENCODING_JITTER**
The default amount of jitter to apply to produced encodings.

Type int

facelift.encode.**DEFAULT_ENCODING_PADDING**
The default padding expected to exist around the detected face frame.

Type float

class facelift.encode.**BaseEncoder**
An abstract encoder class that each encoder should inherit from.

Raises **NotImplementedError** – If the `model_filepath` property is not implemented

get_encoding(*frame*, *face*, *jitter=0*, *padding=0.25*)
Calculate the encoding for a given frame and detected face.

Examples

```
>>> from facelift.capture import iter_media_frames
>>> from facelift.detect import BasicFaceDetector
>>> from facelift.encode import BasicFaceEncoder
>>> detector = BasicFaceDetector()
>>> encoder = BasicFaceEncoder()
>>> for frame in iter_media_frames(MEDIA_FILEPATH):
...     for face in detector.iter_faces(frame):
...         face_encoding = encoder.get_encoding(frame, face)
```

Parameters

- **frame** (*Frame*) – The frame the face was detected in
- **face** (*Face*) – The detected face from the given frame
- **jitter** (*int, optional*) – The amount of jitter to apply during encoding. This can help provide more accurate encodings for frames containing the same face. Defaults to *DEFAULT_ENCODING_JITTER*.
- **padding** (*float, optional*) – The amount of padding to apply to the face frame during encoding. Defaults to *DEFAULT_ENCODING_PADDING*.

Returns The encoding of the provided face for the given frame

Return type Encoding

abstract property model_filepath

Property filepath to the encoding model that should be used for encoding.

Raises **NotImplementedError** – Must be implemented by subclasses

Return type Path

score_encoding(*source_encoding*, *known_encodings*)

Score a source encoding against a list of known encodings.

Important: This score is the average Euclidian distance between the given encodings. So the most similar encodings will result in a score closest to `0.0`.

If no encodings are given, then we will default to using `math.inf` as it is the greatest distance from `0.0` that we can define.

Examples

```
>>> from facelift.capture import iter_media_frames
>>> from facelift.detect import BasicFaceDetector
>>> from facelift.encode import BasicFaceEncoder
>>> detector = BasicFaceDetector()
>>> encoder = BasicFaceEncoder()
>>> # A list of previously encoded faces for a single person
>>> KNOWN_FACES = [...]
>>> for frame in iter_media_frames(MEDIA_FILEPATH):
...     for face in detector.iter_faces(frame):
...         face_encoding = encoder.get_encoding(frame, face)
...         score = encoder.score_encoding(face_encoding, KNOWN_FACES)
```

Parameters

- **source_encoding** (*Encoding*) – The unknown encoding we are attempting to score.
- **known_encodings** (List[*Encoding*]) – A list of known encodings we are scoring against. These encodings should all encodings from a single person's face.

Returns The score of a given encoding against a list of known encodings. This value should be greater than 0.0 (lower is better).

Return type float

class facelift.encode.**BasicFaceEncoder**

Encode faces detected by the *BasicFaceDetector*.

This face encoder *can* handle faces detected by both the *BasicFaceDetector* and the *PartialFaceDetector*. However, you should likely only ever be encoding faces for recognition from the lightest model available (*BasicFaceDetector*).

Important: This encoder **can not** handle faces detected using the *FullFaceDetector*. If we determine we are using a face detected by this detector, the *get_encoding()* method will raise a ValueError.

---

get_encoding(*frame*, *face*, *jitter=0*, *padding=0.25*)

Calculate the encoding for a given frame and detected face.

Examples

```
>>> from facelift.capture import iter_media_frames
>>> from facelift.detect import BasicFaceDetector
>>> from facelift.encode import BasicFaceEncoder
>>> detector = BasicFaceDetector()
>>> encoder = BasicFaceEncoder()
>>> for frame in iter_media_frames(MEDIA_FILEPATH):
...     for face in detector.iter_faces(frame):
...         face_encoding = encoder.get_encoding(frame, face)
```

Parameters

- **frame** (*Frame*) – The frame the face was detected in
- **face** (*Face*) – The detected face from the given frame
- **jitter** (*int, optional*) – The amount of jitter to apply during encoding. This can help provide more accurate encodings for frames containing the same face. Defaults to *DEFAULT_ENCODING_JITTER*.
- **padding** (*float, optional*) – The amount of padding to apply to the face frame during encoding. Defaults to *DEFAULT_ENCODING_PADDING*.

Raises **ValueError** – When the given face was detected with the *FullFaceDetector*.

Returns The encoding of the provided face for the given frame

Return type Encoding

facelift.encode.**get_encoder**(*model_filepath*)

Build an encoder for the given `dlib` ResNet model.

Parameters **model_filepath** (*Path*) – The path to the encoder model

Raises **FileNotFoundError** – If the given model filepath does not exist

Returns The encoder to use for encoding face frames

Return type *Encoder*

8.1.7 facelift.helpers

Contains mechanisms to extract details or normalized details for detected faces.

facelift.helpers.**DEFAULT_NORMALIZED_FACE_SIZE**
The default size of the normalized face frame. Defaults to 256.

Type int

facelift.helpers.**DEFAULT_NORMALIZED_LEFT_EYE_POSITION**
The default percentage (0.0-1.0) where the left eye should be placed in the normalized face frame. Defaults to (0.35, 0.35).

Type Tuple[float, float]

facelift.helpers.**get_eye_angle**(*face*)
Get the angle the eyes are currently at for the given face.

Parameters **face** (*Face*) – The face to get the eye angle from.

Returns The floating point value describing the angle of the eyes in the face.

Return type numpy.float64

facelift.helpers.**get_eye_center_position**(*face*)
Get the center position between the eyes of the given face.

Parameters **face** (*Face*) – The face to extract the center position from.

Returns The position directly between the eyes of the face

Return type Tuple[numpy.int64, numpy.int64]

facelift.helpers.**get_eye_deltas**(*face*)
Get the difference between eye positions of the given face.

Parameters **face** (*Face*) – The face to get the eye deltas from.

Returns A tuple of (x delta, y delta) for the given face's eyes

Return type Tuple[numpy.int64, numpy.int64]

facelift.helpers.**get_eye_distance**(*face*)
Get the distance between the eyes of the given face.

Parameters **face** (*Face*) – The face to get the eye distance from.

Returns A floating point value describing the distance between the face's eye.

Return type numpy.float64

facelift.helpers.**get_eye_positions**(*face*)
Get the center position tuples of eyes from the given face.

Parameters **face** (*Face*) – The face to extract eye positions from.

Raises **ValueError** – If the given face is missing either left or right eye landmarks

Return type Tuple[Tuple[int64, int64], Tuple[int64, int64]]

Returns A tuple of (left eye position, right eye position)

facelift.helpers.**get_normalized_frame**(*frame*, *face*, *desired_width=None*, *desired_height=None*, *desired_left_eye_position=None*)

Get a normalized face frame where the face is aligned, cropped, and positioned.

Examples

Get a normalized face frame from a detected face from the given frame:

```
>>> from facelift.helpers import get_normalized_frame
>>> normalized_frame = get_normalized_frame(frame, face)
```

Parameters

- **frame** (*Frame*) – The original frame the face was detected from.
- **face** (*Face*) – The detected face to use when extracting a normalized face frame.
- **desired_width** (*Optional[int], optional*) – The desired width of the normalized frame. Defaults to None.
- **desired_height** (*Optional[int], optional*) – The desired height of the normalized frame. Defaults to None.
- **desired_left_eye_position** (*Optional[Tuple[float, float]], optional*) – The desired position point for the left eye. This position is a value between 0.0 and 1.0 indicating the percentage of the frame. Defaults to None.

Returns The normalized face frame.

Return type *Frame*

8.1.8 facelift.render

Contains some very basic wrappers around drawing things onto frames.

When detecting faces, it is kinda nice to be able to see what features are being detected and where inaccuracies are being detected. With a combination of the *window* module and some of these helper functions, we can easily visualize what features are being detected.

For example, if we wanted to draw lines for each detected feature from the *PartialFaceDetector* we can do the following:

```
>>> from facelift.capture import iter_stream_frames
>>> from facelift.window import opencv_window
>>> from facelift.detect import PartialFaceDetector
>>> from facelift.render import draw_line
>>> detector = PartialFaceDetector()
>>> with opencv_window() as window:
...     for frame in iter_stream_frames():
...         for face in detector.iter_faces(frame):
...             for _, points in face.landmarks.items():
```

(continues on next page)

(continued from previous page)

```
...                 frame = draw_line(frame, points)
...         window.render(frame)
```

facelift.render.**DEFAULT_COLOR**
The default color for all draw helper functions. Defaults to (255, 255, 255), or white.

Type Tuple[int, int, int]

facelift.render.**DEFAULT_FONT**
The default OpenCV HERSHEY font to use for rendering text. Defaults to `cv2.FONT_HERSHEY_SIMPLEX`

Type int

class facelift.render.**LineType**(*value*)
Enumeration of the different available PointSequence types for OpenCV.

FILLED
Filled line (useful for single points).

CONNECTED_4
A 4-point connected line.

CONNECTED_8
An 8-point connected line.

ANTI_ALIASED
An anti-aliased line (good for drawing curves).

class facelift.render.**Position**(*value*)
Enumeration of available relative positions.

START
Positioned content appears at the left of the container.

END
Positioned content appears at the right of the container.

CENTER
Positioned content appears in the middle of the container.

facelift.render.**draw_contour**(*frame*, *line*, *color=255, 255, 255*, *thickness=-1*, *line_type=cv2.LINE_AA*)
Form and draw a contour for the given line on a frame.

Examples

Draw a contour between multiple points.

```
>>> from facelift.render import draw_contour
>>> frame = draw_contour(frame, [(10, 10), (20, 20)])
```

Parameters

- **frame** (*Frame*) – The frame to draw the contour on.
- **line** (*PointSequence*) – The array of points to use to form the contour.
- **color** (*Tuple[int, int, int], optional*) – The color of the contour.. Defaults to DEFAULT_COLOR.
- **thickness** (*int, optional*) – The thickness of the contour. Defaults to -1.

- **line_type** (*LineType, optional*) – The line type to use for the contour. Defaults to LineType.ANTI_ALIASED.

Return type Type[ndarray]

Returns *Frame* The frame with the contour drawn on it

facelift.render.**draw_line**(*frame*, *line*, *sequence=None*, *color=255, 255, 255*, *thickness=1*, *line_type=cv2.LINE_AA*)

Draw a sequence of connected points on a given frame.

Examples

Draw a line between a sequence of points.

```
>>> from facelift.render import draw_line
>>> frame = draw_line(frame, [(10, 10), (20, 20)])
```

Parameters

- **frame** (*Frame*) – The frame to draw the line on.
- **line** (*PointSequence*) – The array of points to draw on the given frame
- **sequence** (*Optional[List[Tuple[int, int]]], optional*) – An optional custom sequence for drawing the given line points. Defaults to None.
- **color** (*Tuple[int, int, int], optional*) – The color of the line. Defaults to DEFAULT_COLOR.
- **thickness** (*int, optional*) – The thickness of the line. Defaults to 1.
- **line_type** (*LineType, optional*) – The type of the line. Defaults to LineType.FILLED.

Return type Type[ndarray]

Returns *Frame* The frame with the line drawn on it

facelift.render.**draw_point**(*frame*, *point*, *size=1*, *color=255, 255, 255*, *thickness=-1*, *line_type=cv2.FILLED*)

Draw a single point on a given frame.

Examples

Draw a single point a position (10, 10) on a given frame.

```
>>> from facelift.render import draw_point
>>> frame = draw_point(frame, (10, 10))
```

Parameters

- **frame** (*Frame*) – The frame to draw the point
- **point** (*Point*) – The pixel coordinates to draw the point
- **size** (*int, optional*) – The size of the point. Defaults to 1.
- **color** (*Tuple[int, int, int], optional*) – The color of the point. Defaults to DEFAULT_COLOR.

- **thickness** (*int, optional*) – The thickness of the point. Defaults to -1.
- **line_type** (*LineType, optional*) – The type of line type to use for the point. Defaults to LineType.FILLED.

Return type Type[ndarray]

Returns *Frame* The frame with the point drawn on it

facelift.render.**draw_points**(*frame*, *points*, *size=1*, *color=255, 255, 255*, *thickness=-1*, *line_type=cv2.FILLED*)

Draw multiple points on a given frame.

Examples

Draw a sequence of points to a given frame.

```
>>> from facelift.render import draw_points
>>> frame = draw_points(frame, [(10, 10), (20, 20)])
```

Parameters

- **frame** (*Frame*) – The frame to draw the points on.
- **points** (*PointSequence*) – The sequence of points to draw.
- **size** (*int, optional*) – The size of the points. Defaults to 1.
- **color** (*Tuple[int, int, int], optional*) – The color of the points. Defaults to DEFAULT_COLOR.
- **thickness** (*int, optional*) – The thickness of the points. Defaults to -1.
- **line_type** (*LineType, optional*) – The type of line type to use for the points. Defaults to LineType.FILLED.

Return type Type[ndarray]

Returns *Frame* The frame with the points drawn on it

facelift.render.**draw_rectangle**(*frame*, *start*, *end*, *color=255, 255, 255*, *thickness=1*, *line_type=cv2.LINE_AA*)

Draw a rectangle on the given frame.

Examples

Draw a rectangle starting at (10, 10) and ending at (20, 20).

```
>>> from facelift.render import draw_rectangle
>>> frame = draw_rectangle(frame, (10, 10), (20, 20))
```

Parameters

- **frame** (*Frame*) – The frame to draw the rectangle on.
- **start** (*Point*) – The starting point of the rectangle.
- **end** (*Point*) – The ending point of the rectangle.

- **color** (*Tuple[int, int, int], optional*) – The color of the rectangle. Defaults to DEFAULT_COLOR.
- **thickness** (*int, optional*) – The thickness of the rectangle. Defaults to 1.
- **line_type** (LineType, *optional*) – The line type to use when drawing the lines of the rectangle. Defaults to LineType.ANTI_ALIASED.

Return type Type[ndarray]

Returns *Frame* The frame with the rectangle drawn on it

facelift.render.**draw_text**(*frame*, *text*, *start*, *end*, *color=(255, 255, 255)*, *font=cv2.FONT_HERSHEY_SIMPLEX*, *font_scale=1*, *thickness=1*, *line_type=cv2.LINE_AA*, *x_position=<Position.START: 'start'>*, *y_position=<Position.START: 'start'>*, *x_offset=0*, *y_offset=0*, *allow_overflow=False*)

Draw some text on the given frame.

Examples

Draw the text "Hello, World!" right-aligned within the text rectangle from (10, 10) to (20, 20).

```
>>> from facelift.render import draw_text, Position
>>> frame = draw_text(
...     frame,
...     "Hello, World",
...     (10, 10),
...     (20, 20),
...     x_position=Position.END
... )
```

Parameters

- **frame** (*Frame*) – The frame to draw some text on
- **text** (*str*) – The text to draw on the frame
- **start** (*Point*) – The starting point of the text container
- **end** (*Point*) – The ending point of the text container
- **color** (*Tuple[int, int, int], optional*) – The color of the text. Defaults to DEFAULT_COLOR.
- **font** (*int, optional*) – The OpenCV hershey font to draw the text with. Defaults to DEFAULT_FONT.
- **font_scale** (*float, optional*) – The scale of the font. Defaults to 1.
- **thickness** (*int, optional*) – The thickness of the font. Defaults to 1.
- **line_type** (LineType, *optional*) – The line type of the font. Defaults to LineType.ANTI_ALIASED.
- **x_position** (Position, *optional*) – The x-axis position to draw the text in relative to the text container. Defaults to Position.START.
- **y_position** (Position, *optional*) – The y-axis position to draw the text in relative to the text container. Defaults to Position.START.

- **x_offset** (*int, optional*) – The x-axis offset from the text container to add to the calculated relative position. Defaults to 0.
- **y_offset** (*int, optional*) – The y-axis offset from the text container to add to the calculated relative position. Defaults to 0.
- **allow_overflow** (*bool, optional*) – If set to `True`, the provided text will start drawing at the given start and end points without obeying them as a bounding text container. Defaults to False.

Return type `Type[ndarray]`

Returns *Frame* The frame with the text drawn on it

8.1.9 facelift.window

Contains some helper abstractions for OpenCV windows and frame rendering.

This collection of window helpers is just to help standardize and cleanup how to interact with OpenCV window displays. The *opencv_window* context manager is very easy to use for getting a quick window for rendering frames as they are produced.

For example:

```
>>> from pathlib import Path
>>> from facelift.window import opencv_window
>>> from facelift.capture import iter_media_frames
>>> with opencv_window() as window:
...     for frame in iter_media_frames(Path("~/my-file.mp4")):
...         window.render(frame)
```

This context manager will produce a new window for rendering the frames read from `my-file.mp4` and will destroy the window once the context is exited.

I wouldn't recommend using this for any kind of production use; mostly the OpenCV window is just useful for debugging.

`facelift.window.`**DEFAULT_WINDOW_TITLE**
The default OpenCV window title if none is supplied. Defaults to "Facelift".

Type str

`facelift.window.`**DEFAULT_WINDOW_DELAY**
The default number of milliseconds to wait between showing frames. Defaults to 1.

Type int

`facelift.window.`**DEFAULT_WINDOW_STEP_KEY**
The default ASCII key index to use as the step key when step is enabled. Defaults to `0x20` (Space).

Type int

class `facelift.window.`**WindowStyle**
Object namespace of available OpenCV window styles.

DEFAULT
The default OpenCV window style.

Type int

AUTOSIZE
Automatically fit window size on creation.

Type int

GUI_NORMAL
Window with a basic GUI experience.

Type int

GUI_EXPANDED
Window with an expanded GUI experience.

Type int

FULLSCREEN
Window that displays frames fullscreen (full-canvas).

Type int

FREE_RATIO
Window that allows for any window ratio.

Type int

KEEP_RATIO
Window that maintains the original window ratio.

Type int

OPENGL
Window rendered via OpenGL. May not work for some machines and will only work if OpenCV is compiled with GPU support.

Type int

class facelift.window.**opencv_window**(*title='Facelift'*, *style=cv2.WINDOW_NORMAL*, *delay=1*, *step=False*, *step_key=32*)
Create an OpenCV window that closes once the context exits.

Examples

Easy usage of OpenCV's provided window to display read frames from a webcam.

```
>>> from facelift.window import opencv_window
>>> with opencv_window() as window:
...     for frame in iter_stream_frames():
...         window.render(frame)
```

Parameters

- **title** (*str*) – The title of the OpenCV window.
- **style** (*int*) – The style of the OpenCV window.
- **delay** (*float*) – The number of milliseconds to delay between displaying frames.
- **step** (*bool*) – Flag that indicates if the window should wait for a press of the defined `step_key` before releasing the render call. Defaults to False.
- **step_key** (*int*) – The ASCII integer index of the key to wait for press when `step` is True. Defaults to `0x20` (Space).

Raises

- **ValueError** – If the given window title is an empty string
- **ValueError** – If the given window delay is less or equal to 0

__enter__()
Initialize the context of the window.

__exit__(*exc_type*, *exc_value*, *traceback*)
Destroy the context of the window.

Parameters

- **exc_type** (Optional[Type[BaseException]]) –
- **exc_value** (Optional[BaseException]) –
- **traceback** (Optional[traceback]) –

Return type Optional[bool]

close()
Destroy the window with the current context's title.

create()
Create a new window with the current context's title and style.

render(*frame*)
Render a given frame in the current window.

Parameters **frame** (*Frame*) – The frame to render within the window

---

8.1.10 facelift._data

Helpers for fetching the pre-trained models this project is built around.

Due to the size of the models that we are building this project around, we need to fetch the models outside of the standard PyPi installation. The following methods handle building an asset manifest that should be released with each GitHub release. This asset manifest will then further inform the little downloading script we have provided where to find and place the assets in the installed package.

This helper utility currently expects the following of the GitHub release:

1. A `data-manifest.json` is provided as a GitHub release asset.
2. All models within the asset manifest are included as GitHub release assets.

Important: The `data-manifest.json` must following the following structure:

```
{
    "relative filepath from package root for asset": [
        "download url of asset",
        "md5 hash of asset"
    ]
}
```

As an example:

```
{
    "data/encoders/dlib_face_recognition_resnet_model_v1.dat": [
        "https://github.com/stephen-bunn/facelift/releases/download/v0.1.0/dlib_face_
↪recognition_resnet_model_v1.dat",
        "2316b25ae80acf4ad9b620b00071c423"
    ]
}
```

Examples

```
>>> from facelift._data import download_data
>>> download_data(display_progress=True)
https://... [123 / 456] 26.97%
Downloaded https://... to ./... (1234567890)
```

facelift._data.**build_manifest**(*release_tag*, **asset_filepaths*)

Build the manifest content for a proposed release and defined assets.

Parameters

- **release_tag** (*str*) – The release tag the manifest is being built for.
- **asset_filepaths** (pathlib.Path) – Multiple existing local asset filepaths.

Raises

- **FileNotFoundError** – When a given asset filepath does not exist.
- **ValueError** – When a checksum cannot be calculated for one of the given filepaths.

Returns The manifest JSON-serializable dictionary

Return type Dict[str, Tuple[str, str]]

facelift._data.**download_data**(*display_progress=False*, *release_tag=None*, *chunk_size=4096*, *validate=True*)

Download the data from a fetched remote release manifest.

Parameters

- **display_progress** (*bool, optional*) – Flag that indicates if you want to display the download progress for assets. Defaults to False.
- **release_tag** (*Optional[str], optional*) – The release tag of the assets you want to download. Defaults to None which will fetch the latest release assets.
- **chunk_size** (*int, optional*) – The chunk size to use when downloading assets. Defaults to DOWNLOAD_CHUNK_SIZE.
- **validate** (*bool, optional*) – If False, will skip checksum validation for all downloaded assets. Defaults to True.

Raises

- **FileExistsError** – If a file already exists at one of the assets relative file locations.
- **ValueError** – If the downloaded assets fails checksum validation.

facelift._data.**get_remote_manifest**(*release_tag=None*)

Get the manifest content from a GitHub release.

Parameters **release_tag** (*Optional[str], optional*) – The release tag of the manifest to fetch. Defaults to None which fetches the latest release manifest.

Returns The manifest JSON-serializable dictionary

Return type Dict[str, Tuple[str, str]]

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

T

V

W