

# Custom Python Scripts for the Limelight 3A Camera

---

# Introduction

---

Until the 2024 – 2025 IntoTheDeep season, FIRST never allowed any sort of coprocessor for FTC. But that changed with the endorsement of the Limelight 3A camera as the only “programmable vision coprocessor”.

The Limelight will be part of the core architecture in the Mobile Robot Controller that is coming in the 2027 – 2028 season. See <https://community.firstinspires.org/introducing-the-future-mobile-robot-controller>.

# Introduction (2)

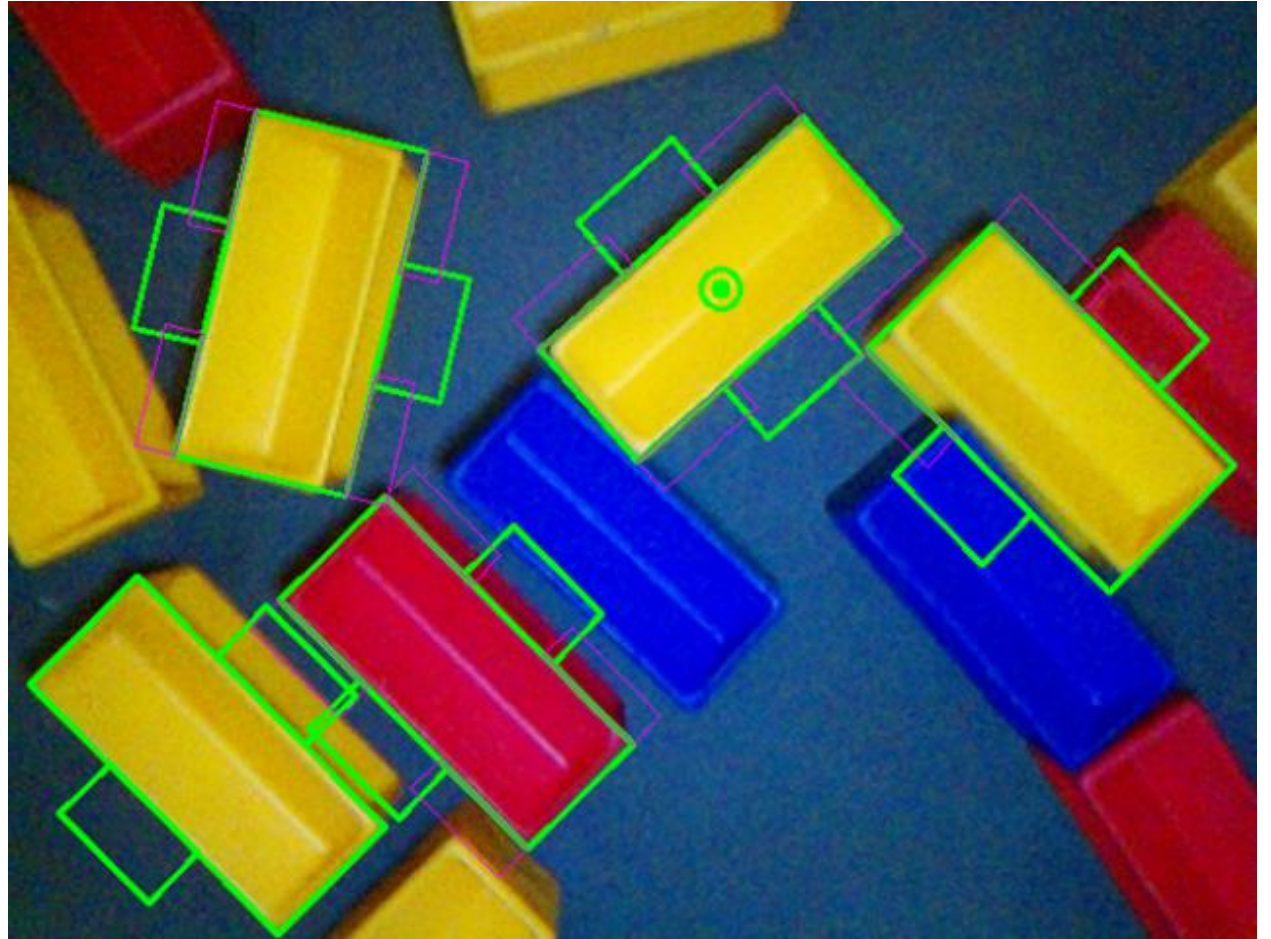
---

What are the advantages of using a Limelight camera?

- Faster image processing than you can do in Java on the Robot Controller
- Built-in features such as AprilTag recognition
- Support for custom Python scripts – the focus of today

The next slide shows an example of what you can do with a custom Python script.

This is an image taken from a Limelight 3A mounted on Notre Dame Team 4348's robot. The green outlines were added during debugging to show the possible "pickup zones" for the robot's intake rollers. The dot in the center of the yellow sample indicates that it is the most centrally located red alliance or neutral sample with a clear pickup zone.



# Topics for today

---

## **Note before we get started on the details**

This presentation and demonstration Python scripts are available on Github at <https://github.com/NDHSRK/Limelight5921>.

The repository also includes a sample FTC Autonomous OpMode that communicates with the Limelight script. Just copy/paste the folder limelightauto into a convenient place in the TeamCode directory of your Android Studio project.

Please contact me, Philip Young, with any questions at [young@ndhs.org](mailto:young@ndhs.org).

# Topics for today (2)

---

We will cover five topics:

1. Simple OpenCV workflow in Python to use as a sample
2. How to develop and test your Python code before uploading it to the Limelight
3. How to use the Limelight's web interface to upload your code and run preliminary tests
4. How to use the Limelight from an FTC Autonomous OpMode
5. What to do when you run into problems



## Basic OpenCV workflow in your Python script

We'll talk about the OpenCV operation called “thresholding” as it applies to color images converted to the HSV color space or to grayscale.



To the right is the grayscale version of the original image with the yellow sample thresholded.

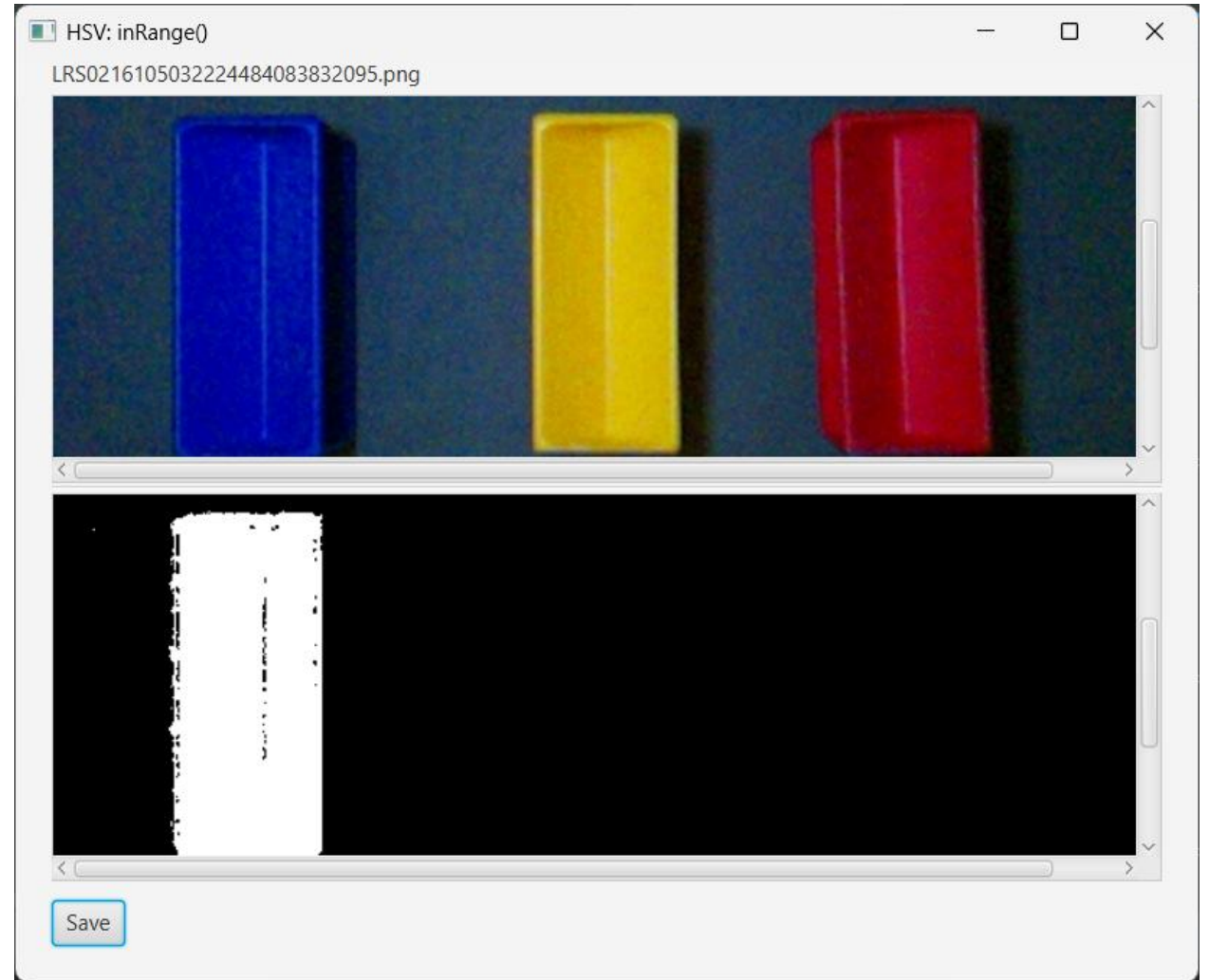
Straight grayscale thresholding of blue or red would not work because of the lack of contrast.



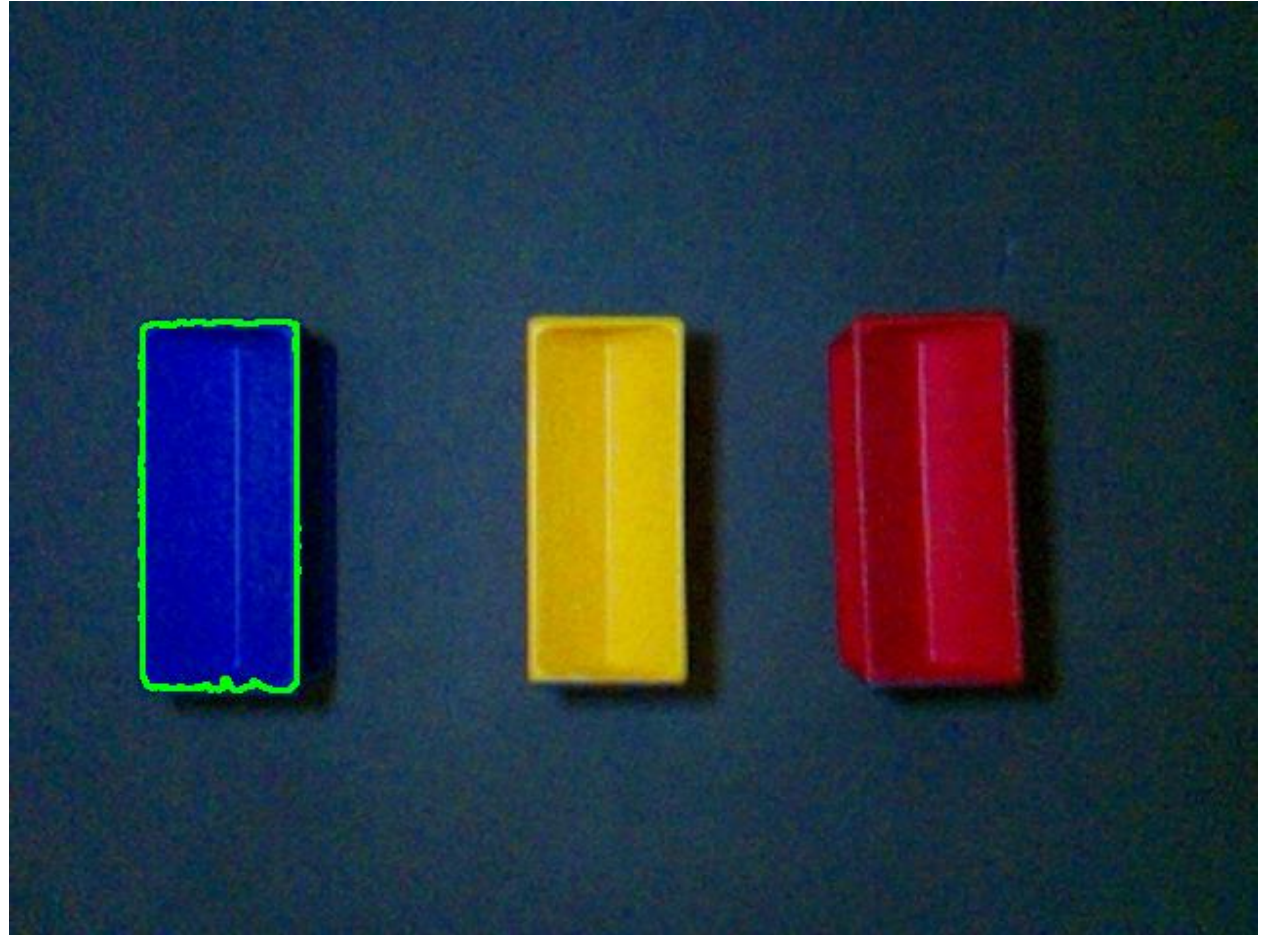


For HSV images the OpenCV thresholding function is called `inRange()`. Here you can get an introduction to HSV with a Python example:

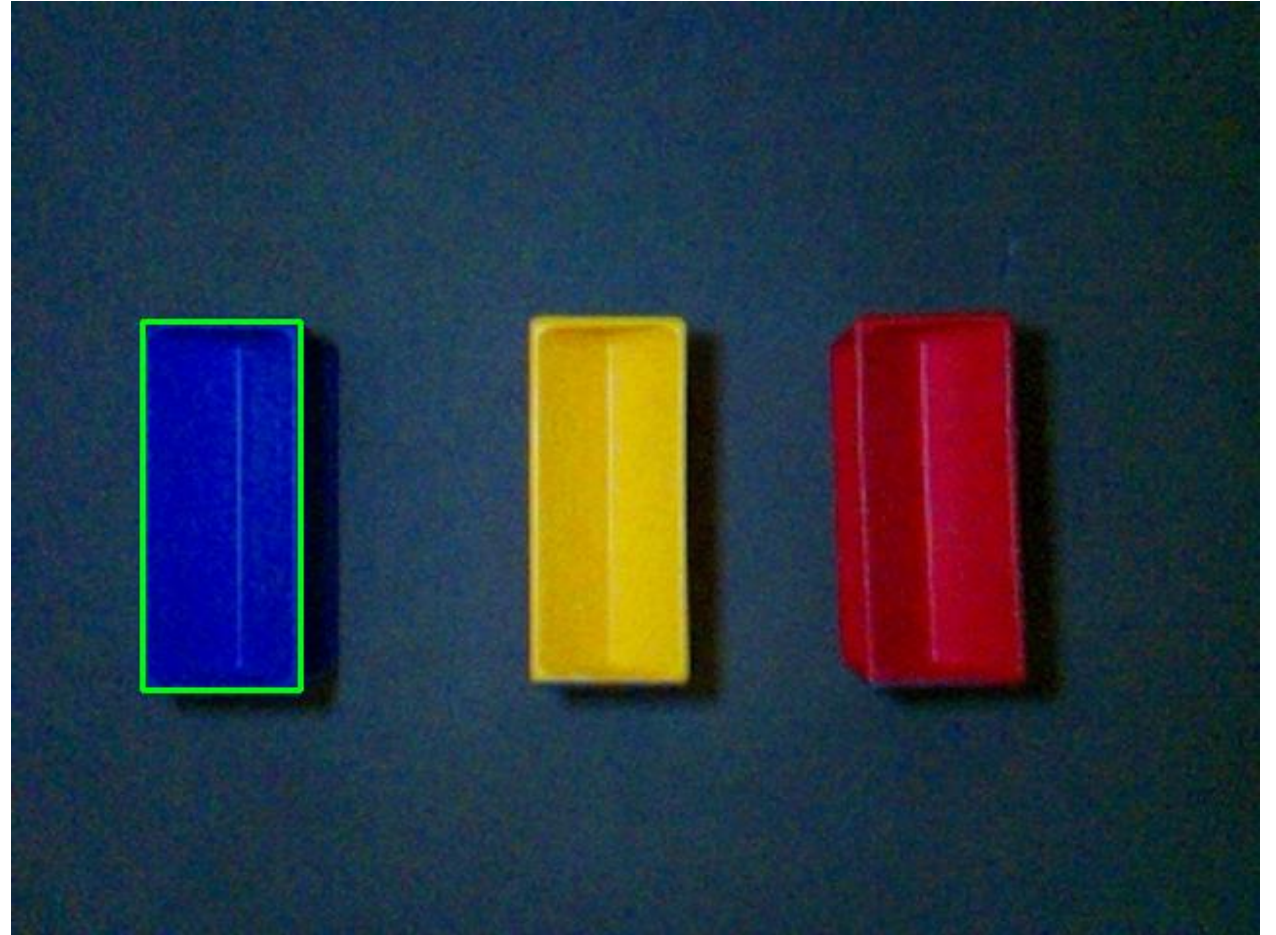
[https://docs.opencv.org/3.4/da/d97/tutorial\\_threshold\\_inRange.html](https://docs.opencv.org/3.4/da/d97/tutorial_threshold_inRange.html)



On the right is the output of a call to `findContours()`, which takes a thresholded binary image as input, that is, the output of grayscale `cv2.threshold()` or HSV `cv2.inRange()`.



On the right is the output of a call to `minAreaRect()`, which takes a single contour from the output of `findContours()` as input and returns an OpenCV `RotatedRect`. The difference between this slide and the previous one doesn't appear to be much, but it's very significant. From a `RotatedRect` you can get the angle of the object.



# How to develop and test your Python code

---

At some point you'll want to upload Python code to the Limelight. But the Limelight itself provides only a basic Python editor, not an Integrated Development Environment or IDE.

PyCharm, which is in the JetBrains family of IDEs just like Android Studio, is a good choice for developing and testing your Python. You can set breakpoints for interactive debugging and you can add calls to `cv2.imwrite` and `cv2.imshow` to write out or display intermediate images.

# Developing your Python code in PyCharm

---

As input to your PyCharm script you need a snapshot image from the Limelight. In the next section we'll see how to save and download a snapshot from the Limelight's web interface.

See Appendix A for guidelines on how to set up PyCharm and the demonstration project.

# How to use the Limelight's web interface to upload your code and run preliminary tests

---

Connect your Limelight to a USB-3 port on a Windows PC and access the web interface at <http://limelight.local:5801/>

Copy and paste your code from PyCharm into the Python editor.

There are some conventions and restrictions ...



# The Python Editor

---

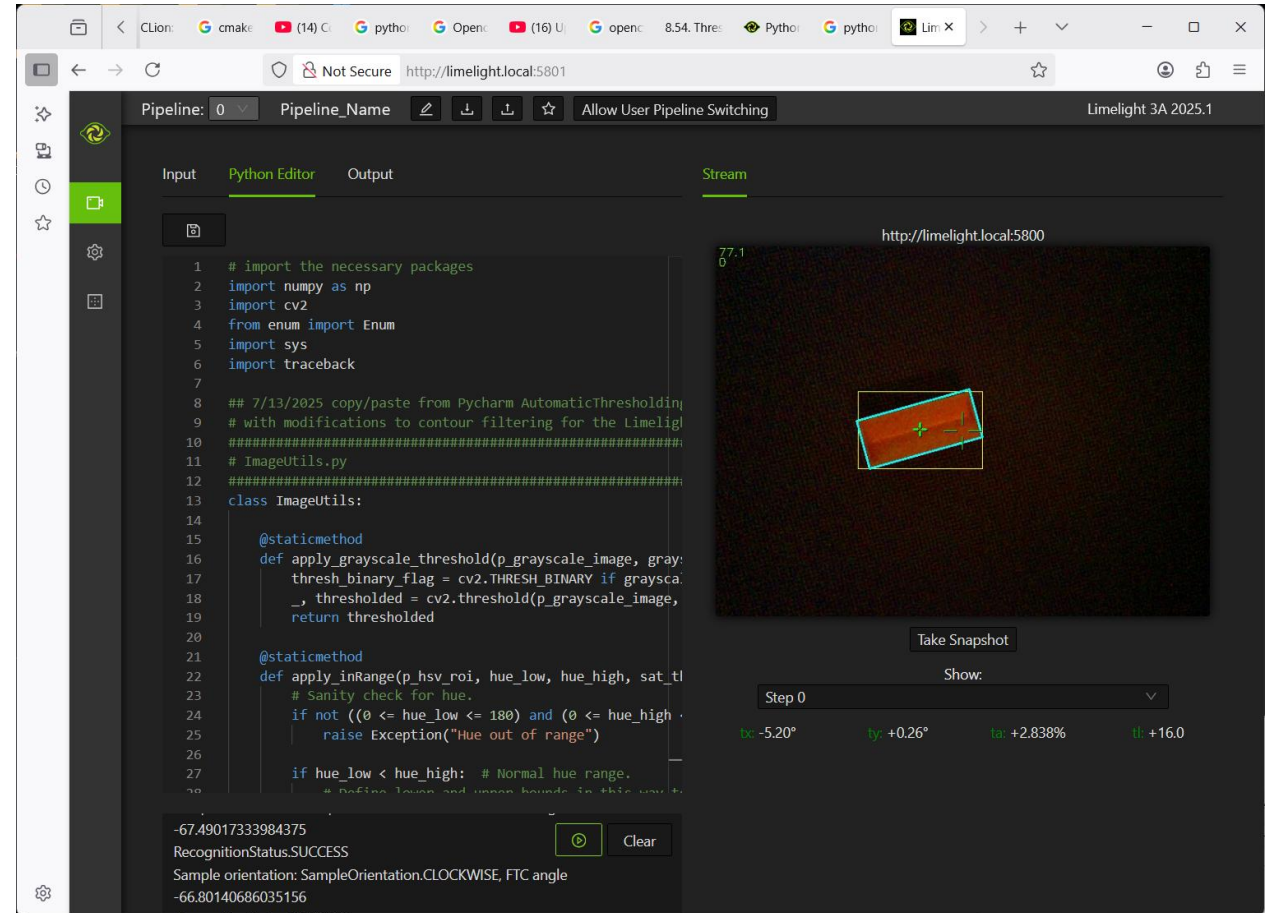
All Python code must be combined into a single script.

The script may not access the local file system on the Limelight, e.g. via calls to `cv2.imwrite`, or Limelight's web UI, e.g. via calls to `cv2.imshow` --- or your Limelight pipeline will crash and most likely lock up the Limelight.

So you *\*must\** remember to remove calls to `cv2.imwrite` and `cv2.imshow` or wrap them in a conditional such as `if platform.system() == "Windows":` before uploading your script to the Limelight.

The output of `platform.system()` on the Limelight is "Linux".

The Python Editor. The Limelight requires that you supply a function called runPipeline. The Limelight runtime calls this function continually and supplies two arguments: a BGR image and llrobot, an array of doubles, the contents of which you determine.



# The Python Editor (2)

---

You *\*must\** temporarily hardcode any llrobot input parameters that you want to test via the web interface because both the test pipeline in PyCharm (runPipeline\_Tester.py) and your Autonomous OpMode supply llrobot but it's not available in the Limelight web configuration.

Let's look at an example.

```
class Alliance(Enum):
    BLUE = 1
    RED = 2

def runPipeline(image, llrobot):
    alliance_ordinal = int(llrobot[0])
    match alliance_ordinal:
        case 1:
            alliance = SampleRecognition.Alliance.BLUE
        case 2:
            alliance = SampleRecognition.Alliance.RED
        case _:
            return np.array([[[]]]), image, [
                SampleRecognition.SampleRecognitionReturn.RecognitionStatus.IDLE.value,
                SampleRecognition.SampleColor.NONE.value]
```

# The Python Editor (3)

---

- To test the red alliance in the web interface, the line –  
*alliance\_ordinal = int(llrobot[0])*  
becomes  
*alliance\_ordinal = 2 # int(llrobot[0])*
- Click the Save icon right above line 1 in the Python Editor window.

# The Python Editor (4)

---

Python scripts run continually on the Limelight; your script determines what is displayed in the right-hand window.


In the lower left of the next screenshot you can see the output of any `print()` statements you've included in your script.



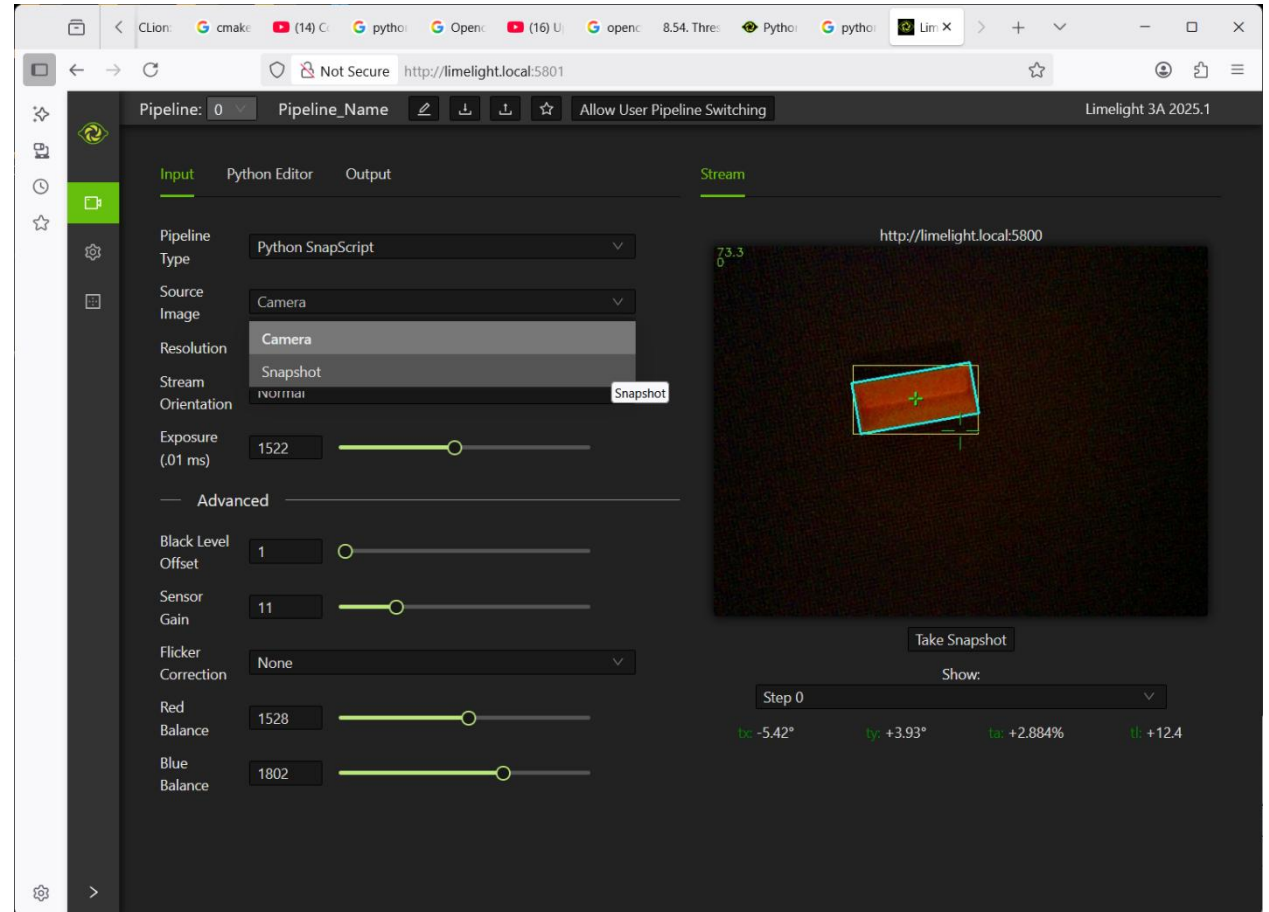
Python print() output in the  
Limelight Python Editor.

```
21     @staticmethod
22     def apply_inRange(p_hsv_roi, hue_low, hue_high, sat_low, sat_high):
23         # Sanity check for hue.
24         if not ((0 <= hue_low <= 180) and (0 <= hue_high <= 180)):
25             raise Exception("Hue out of range")
26
27         if hue_low < hue_high: # Normal hue range.
28             # Define lower and upper bounds in this way
29             # so that we can handle the wrap-around case.
30             lower = hue_low
31             upper = hue_high
32         else: # hue_low > hue_high, wrap-around case.
33             lower = hue_low
34             upper = 360 + hue_high
35
36         # Apply the hue range filter.
37         for i in range(p_hsv_roi.shape[0]):
38             for j in range(p_hsv_roi.shape[1]):
39                 hue = p_hsv_roi[i, j, 0]
40                 if hue < lower or hue > upper:
41                     p_hsv_roi[i, j, 0] = 0
42                     p_hsv_roi[i, j, 1] = 0
43                     p_hsv_roi[i, j, 2] = 0
44
45         # Apply the saturation range filter.
46         for i in range(p_hsv_roi.shape[0]):
47             for j in range(p_hsv_roi.shape[1]):
48                 sat = p_hsv_roi[i, j, 1]
49                 if sat < sat_low or sat > sat_high:
50                     p_hsv_roi[i, j, 0] = 0
51                     p_hsv_roi[i, j, 1] = 0
52                     p_hsv_roi[i, j, 2] = 0
53
54         return p_hsv_roi
```

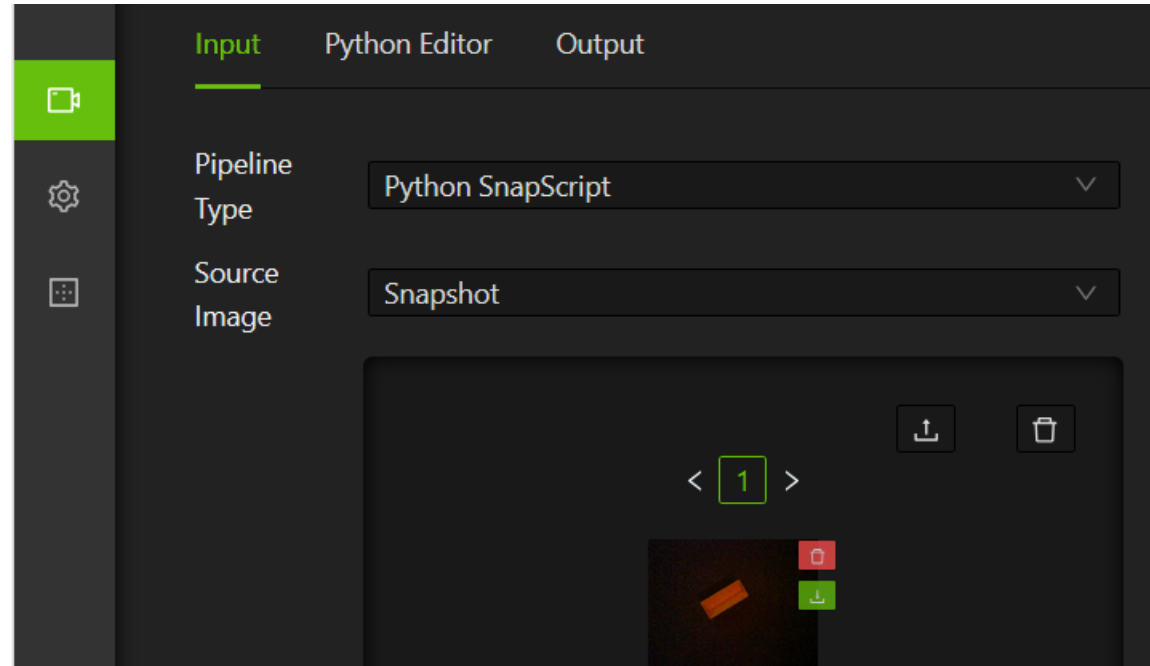
-67.49017333984375  
RecognitionStatus.SUCCESS  
Sample orientation: SampleOrientation.CLOCKWISE, FTC angle  
-66.80140686035156  
RecognitionStatus.SUCCESS

 Clear

On the Limelight's Input web page you choose whether your script receives a stream of images from the camera or whether it runs repeatedly with a saved snapshot.



On either the Input or Python Editor page you can take a snapshot, which the Limelight saves into its file system. You can download the snapshot to your PC by clicking the small download icon next to the image and then use the snapshot as input to your PyCharm project.



# Return values from your script

---

Your runPipeline function returns three values that look like this —

# return a contour for the LL crosshair, the modified image,

# and custom robot data.

*return* aContour, image, llpython

# Return values from your script (2)

---

Argument 0; an OpenCV contour or the empty argument `np.array([])`.

Argument 1: a BGR image (typically one onto which you've drawn debugging information), or the unchanged input image.

Argument 2: your lpython array of doubles.

# Return values from your script (3)

---

Where do the return values go?

The Limelight uses the contour (argument 0) to set the crosshair positions. Team 4348 preferred to use IIPython to send the x and y positions of the target sample as well as its FTC angle back to the Autonomous OpMode.

The Limelight displays the BGR image (argument 1) in the right-hand panel of the Python Editor web page. This image is *\*not\** passed back to your Autonomous OpMode.

Only IIPython, which you determine, is returned to your Autonomous OpMode.



# How to use the Limelight from an Autonomous OpMode

---

Simple workflow:

1. Initialize the Limelight
2. Encode and send llobot to your Python script on the Limelight
3. Poll the Limelight for results
4. Decode lpython and act on the results

Input to the Limelight: llobot – an array of up to 8 doubles

Output from the Limelight: lpython – an array of up to 32 doubles

# Terminology

---

***Serialization*** is the process of converting an object's state into a stream of bytes, while ***marshaling*** is the broader process of preparing data for transmission or storage, which may include serialization as a step.

The Limelight API on both sides of the wire, the FTC side and the Limelight camera side, takes care of **serialization**. But you must be concerned with **marshaling**.

# How to marshal llrobot

---

This example shows how to send an alliance value from Java to Python. On the Java side you have this declaration:

```
public enum Alliance {  
    NONE, BLUE, RED  
}
```

And on the Python side you have an exactly corresponding Enum. The assigned values match the ordinals of the Java enum.

```
class Alliance(Enum):  
    NONE = 0  
    BLUE = 1  
    RED = 2
```

# Sequence of events in your Autonomous OpMode

---

1. Before `waitForStart()` in your Autonomous OpMode start the Limelight but run an Idle script that does nothing but return an empty value; do this now so that you don't have to wait for the Limelight to start up later
2. After `waitForStart()`, when your OpMode needs to do image recognition, send the Java Enum value for your alliance to the Limelight by getting its integer ordinal and marshalling it, for example, as: `double[] llRobot = {(double) Alliance.RED.ordinal()};`

# Sequence of events in your Autonomous OpMode (2)

---

3. The Limelight API on the FTC side handles the serialization and presents your Python script with an array of doubles. By agreement, unmarshal the first double as an int and create a corresponding Python Enum value.
4. Your Python script performs image recognition using OpenCV libraries that are built into the Limelight.
5. In parallel your Autonomous OpMode polls the Limelight for results.

# Sequence of events in your Autonomous OpMode (3)

---

6. When the Python script is done on the Limelight it marshals the return value as an array of up to 32 doubles. You decide what information to include such as the recognition status (success, failure, Python crash, etc.) and, for a recognition success, the coordinates and angle of the sample.

7. Your Autonomous OpMode receives and unmarshals the array of doubles from the Limelight and acts on the results.



# Requesting a snapshot from the Limelight FTC SDK API

---

There is a function in the Limelight API that requests the Limelight to take a snapshot and store it on the Limelight, `captureSnapshot()`. This can be useful for debugging but to download the snapshot you have to connect the Limelight to a PC and use the web interface.

# How to debug problems that occur during the running of your Autonomous OpMode

---

As an FTA I saw, on a number of occasions, a robot drive up to the submersible after delivering four samples to the basket - and stop. I could only guess that their Limelight script either crashed or failed to recognize a sample.

See Appendix B for guidance on debugging.

# Python errors can crash your script

---

Remember: Python is an interpreted language so even simple errors can cause a crash at runtime.

```
alliance_ordinal = int(llrobot[0])  
print("Alliance ordinal " + alliance_ordinal)
```

As a corollary: part of the above statement is underlined in PyCharm as a warning – so always pay attention to warnings!

Not to mention indentation errors!

# Twists and turns with the Limelight

---

Note the difference in image quality between the Limelight and the Logitech C920 webcam in the two images above.

If you use the Limelight you will have to be aware of and constantly adjust to lighting.

This topic is also addressed in Appendix B.

# Where to start with OpenCV

---

No matter what your background --

- You know some Python but are new to OpenCV
- You know some OpenCV in Java or c++ but you're new to Python
- You're new to both Python and OpenCV

You will want to start here: **[pyimagesearch.com](http://pyimagesearch.com)**. The collection of articles on this website offers excellent examples and comprehensive explanations of how to use OpenCV in Python.

# Appendix A: Setting up PyCharm

---

First make sure you have a Python interpreter; if not, download the latest from Python.org.

Download the free community edition of PyCharm from JetBrains:  
<https://www.jetbrains.com/pycharm/download/?section=windows>

Download the zip file for the project from  
<https://github.com/NDHSRK/Limelight5921>.

Right click the zip file and select “Extract all”.

Move the Limelight5921 PyCharm project folder into your directory tree.

# Appendix A: Setting up PyCharm (2)

---

Open PyCharm, select “Open” from the menu at the top, and navigate to place in your directory tree where you moved the Limelight5921 project.

Look at the README.md file in the project for further detailed instructions.

# Appendix B: Debugging

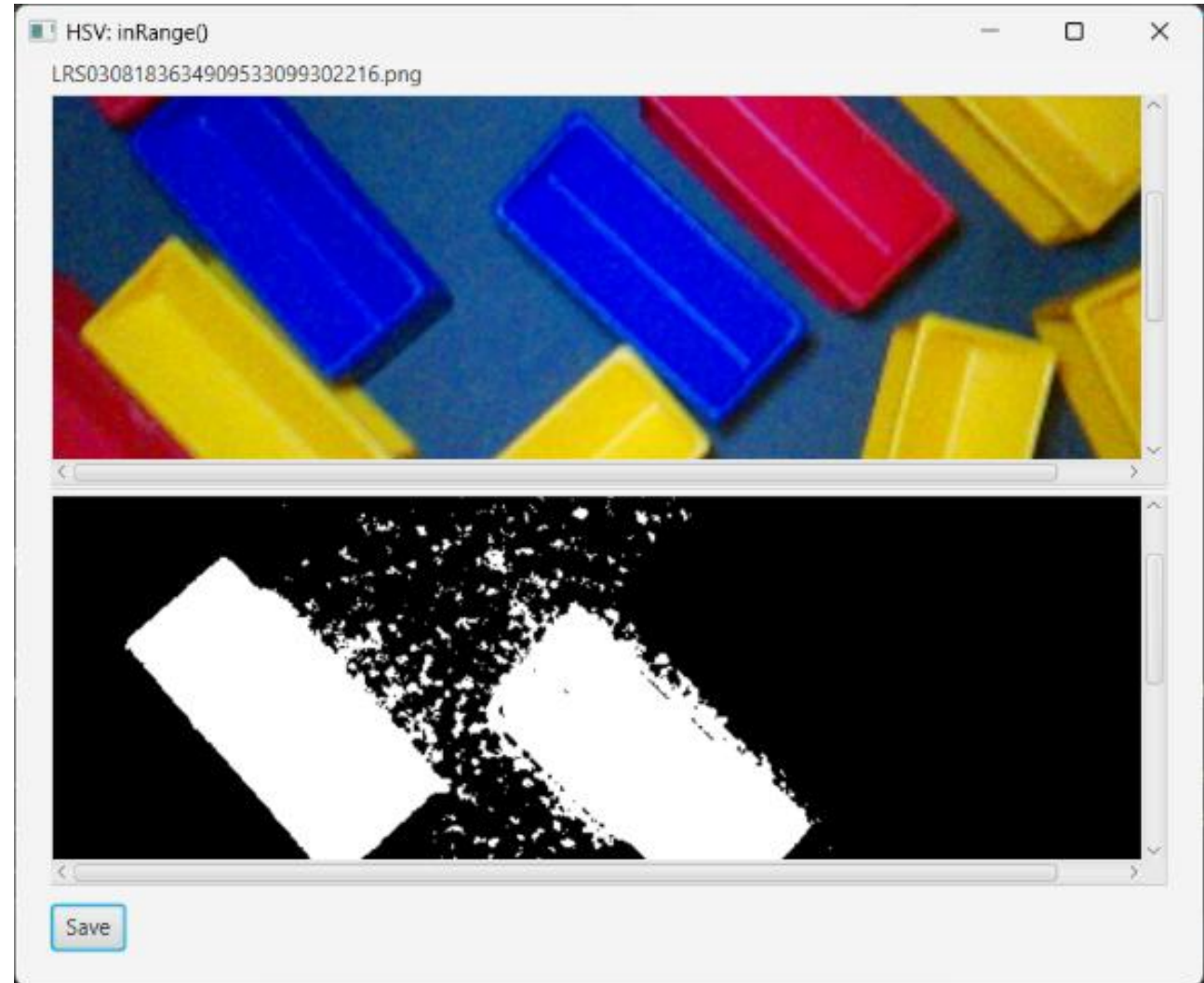
---

Check the FTC matchlogs first for any log entries that your Autonomous OpMode has written out.

Do you know how to write RobotLog entries and where to find the matchlogs?



It can be hard to tell when different lighting conditions cause problems with thresholding.



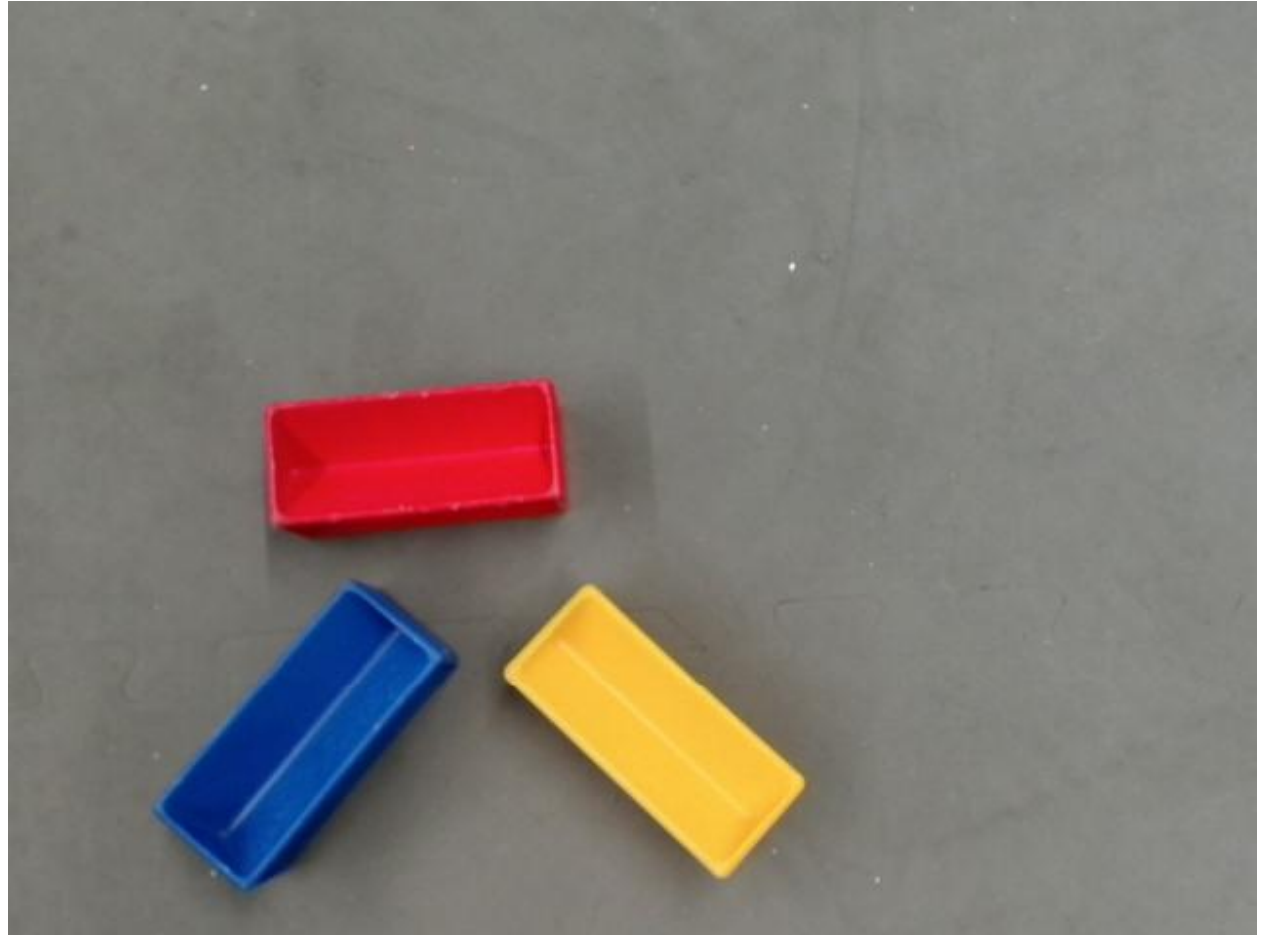
# Thresholding problems

---

First you have to recognize that you have a problem. In the IIPython return from our Limelight script Team 4348 included the number of detected contours. With a busy image such as the above, this number can be in the hundreds!

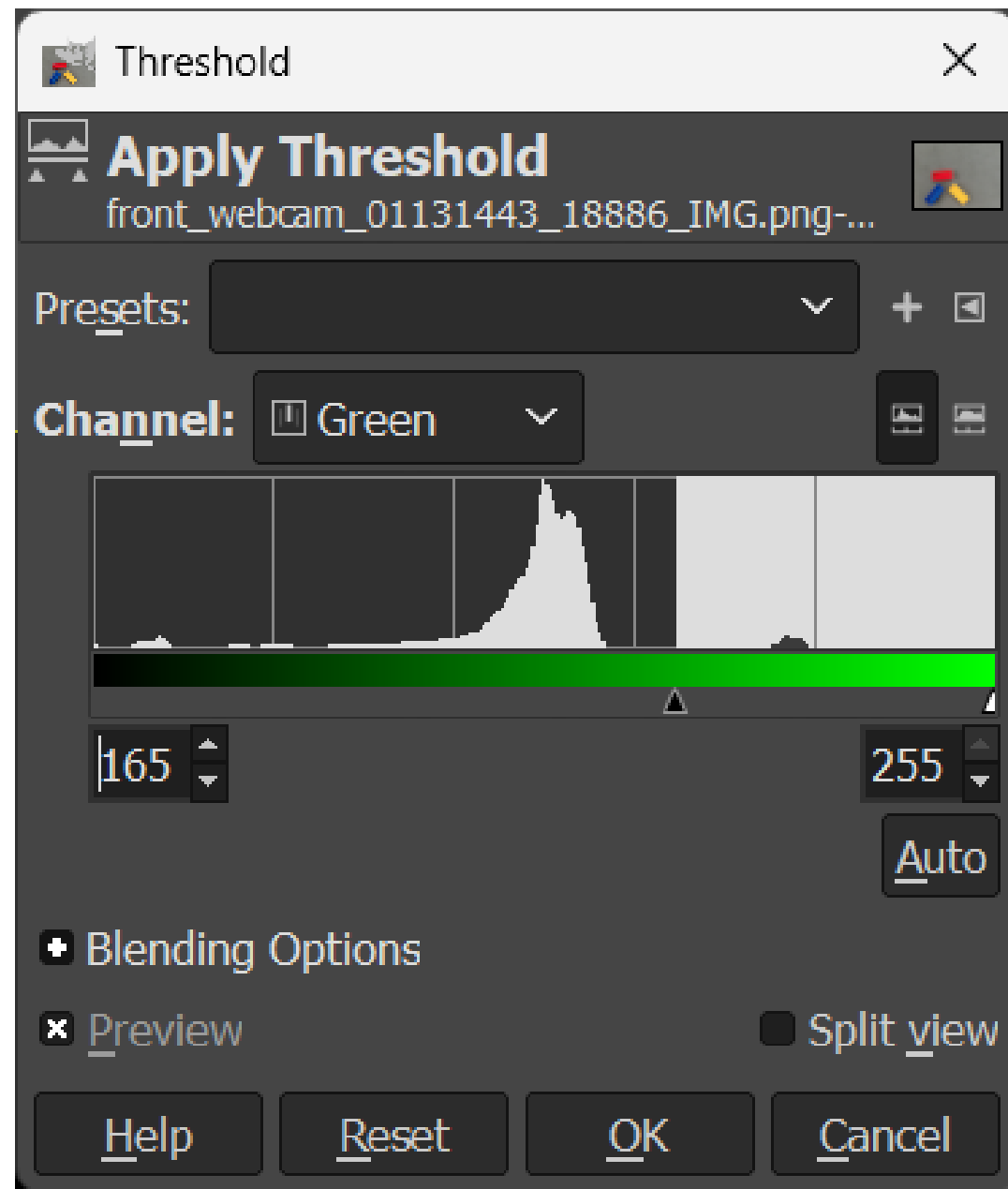
You need to get a snapshot from the Limelight that you can use as input to your PyCharm script where you can debug your script interactively. It can be helpful to use Gimp to dial in the correct thresholding values for your lighting conditions.

Let's start with a different example, this one from the Logitech C920 webcam.

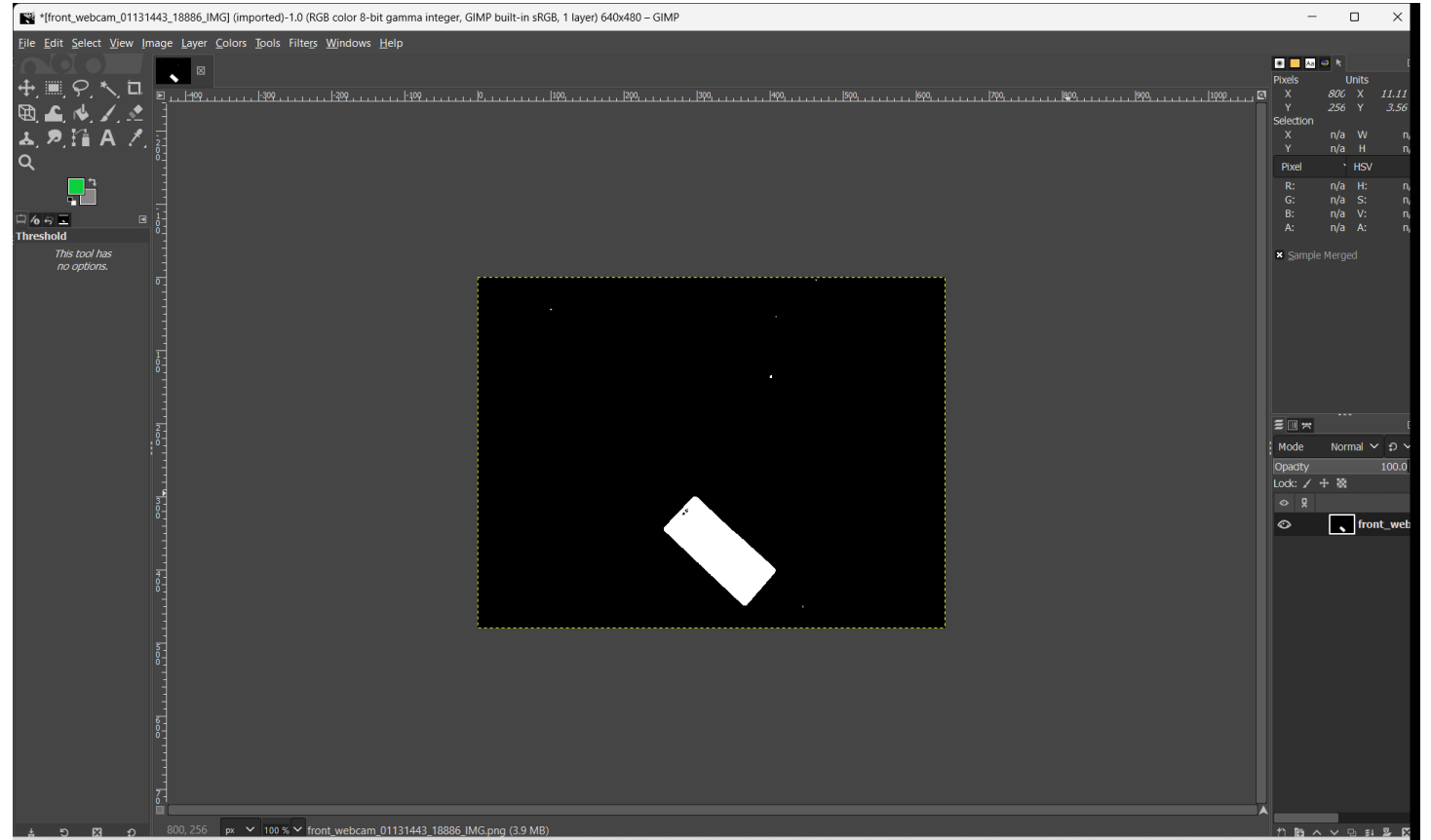


This is the Gimp thresholding menu for grayscale. You can see that the low threshold value is set to 165; any pixel with a greater grayscale value is set to 255 (white).

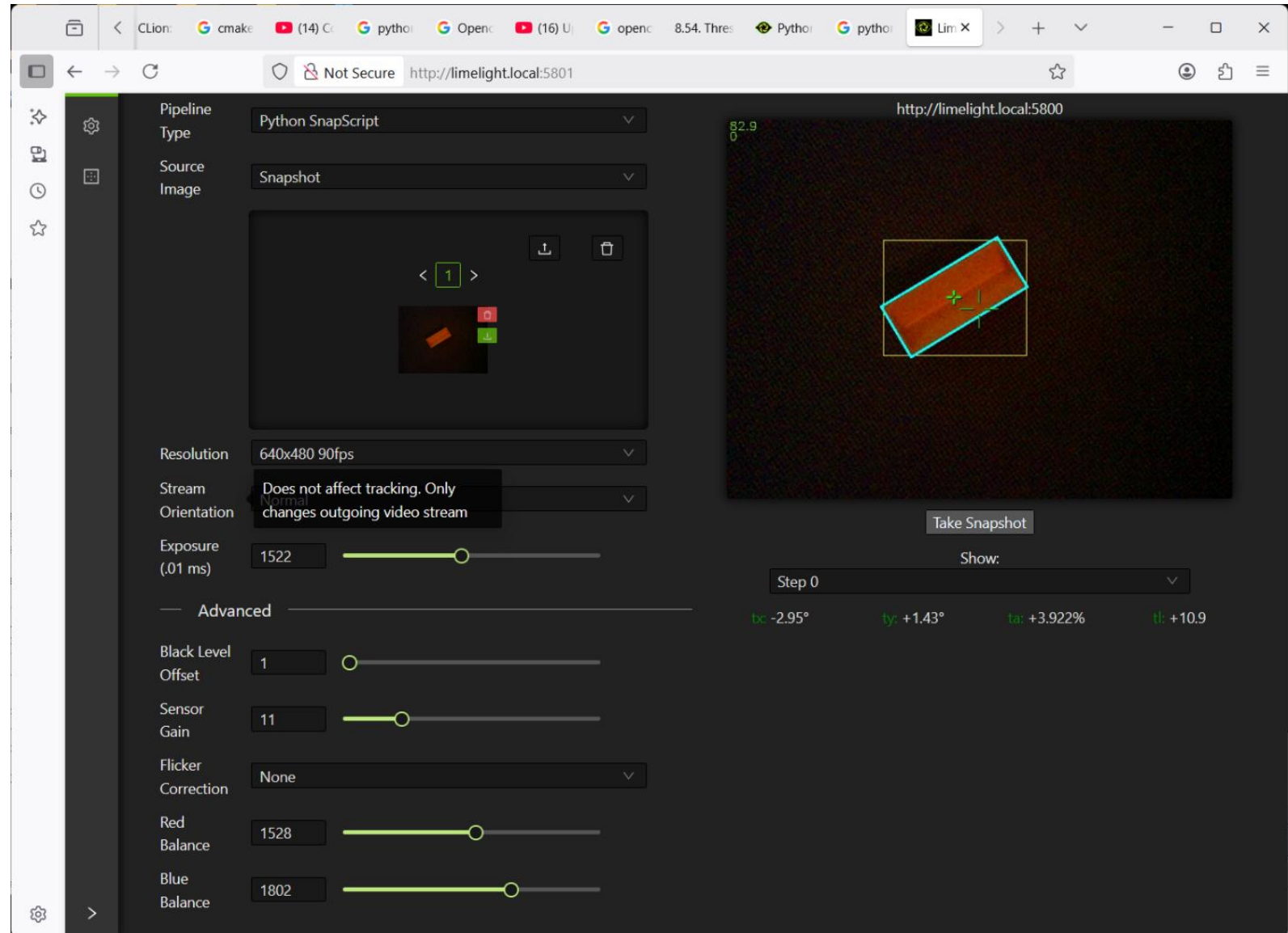
Specifying the parameters for HSV is more complicated.



And here is the output from  
Gimp thresholding.



The Limelight's Input settings can directly affect the quality of the image that the Limelight presents to your script.



# Limelight input settings

---

You'll have to experiment with --

- Exposure
- Sensor gain
- Red and blue balance