Image Basics

Image Basics

This section will cover :

- Digital Images
- Colour Spaces
- Reading and Writing Images
- Showing Images
- Pixels
- Drawing on Images
- User Input

These topics will be **implemented** and **tested** in Python with OpenCV.

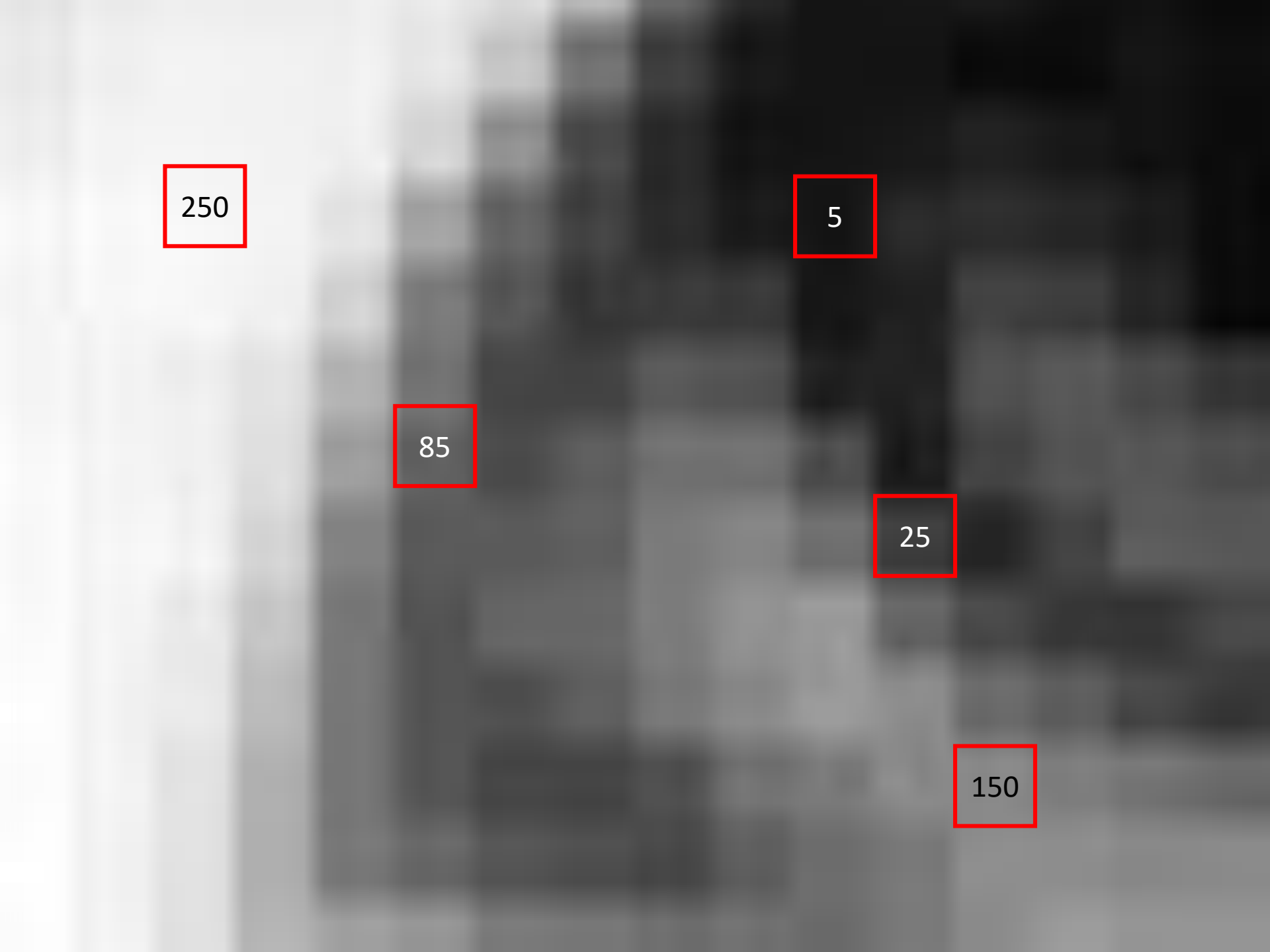
Digital Images

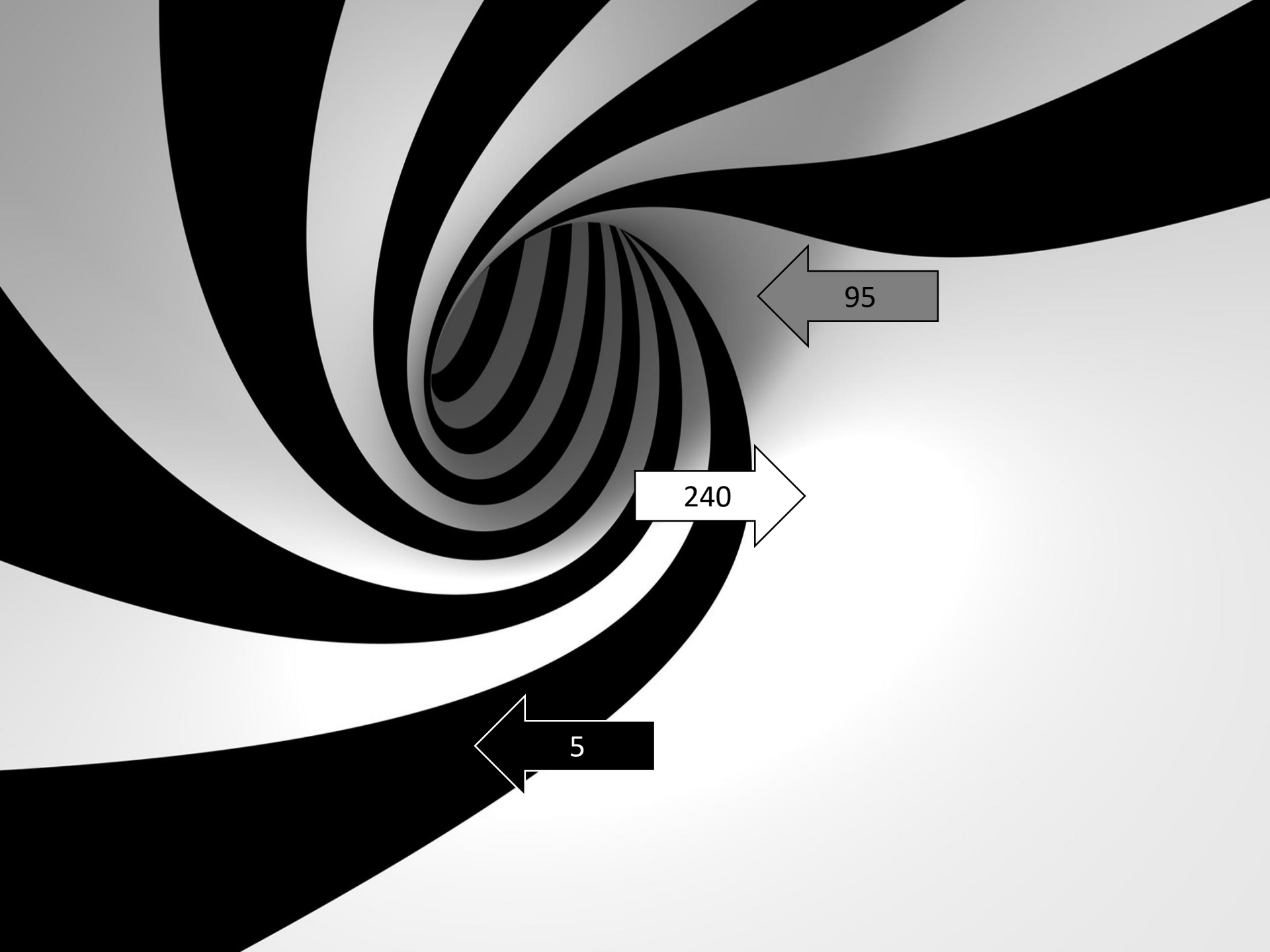
Digital Images

Images are originally captured as light and converted to a signal. These signals are then broken down into discrete *pixels* (sampled), creating digital images.
Capture technologies can vary from photographic, to X-Ray, to infra-red and beyond.

Digital Images

The first digital images, called *dot matrix* images, were produced in the early 1920s and introduced the predecessor of the pixel


250
5
85
25
150

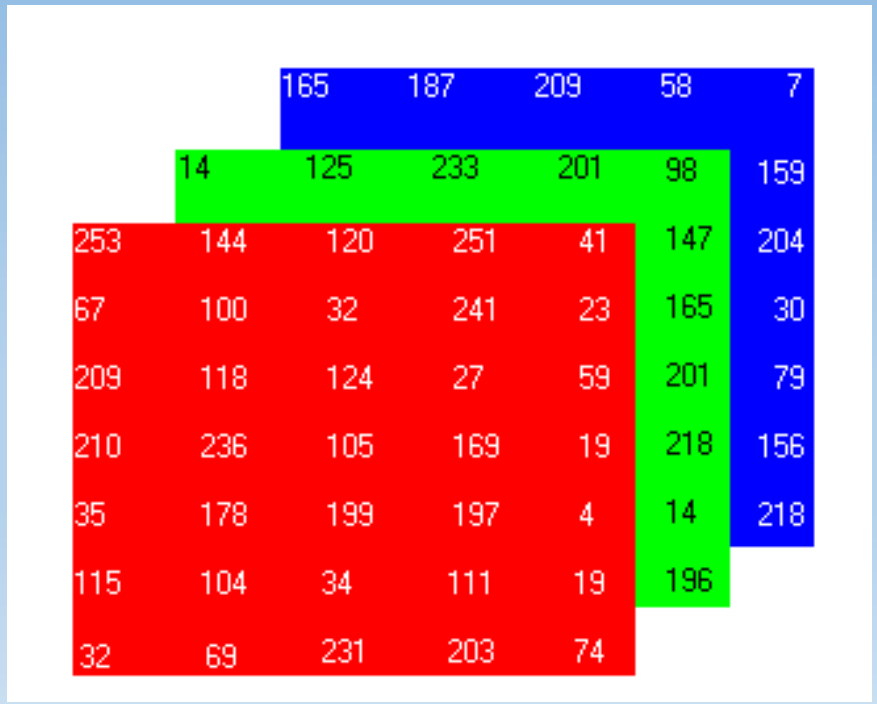

95
240
5


RGB = 5, 240, 5
RGB = 240, 240, 5
RGB = 240, 5, 240

Digital Images

Images are actually matrices.
Each element (pixel) represents the colour or brightness at that location.
An 8-bit (or 24-bit for colour) image will have values from 0 to 255.
A *grayscale* image has 1 value per pixel, a colour image has 3:

253	144	120	251	41
67	100	32	241	23
209	118	124	27	59
210	236	105	169	19
35	178	199	197	4
115	104	34	111	19
32	69	231	203	74



Colour Spaces

Colour Spaces

The value that is used to represent the pixels will help describe the visual appearance of the image at that point.

We have already seen two examples of pixel representation – grayscale (2D) and RGB (3D).

There are many other ways to represent a pixel, in different *colour spaces*.

Colour Spaces

The main colour spaces which we will investigate here are…

- RGB
- HSV
- YUV
- Grayscale
- Binary

Colour Spaces

RGB

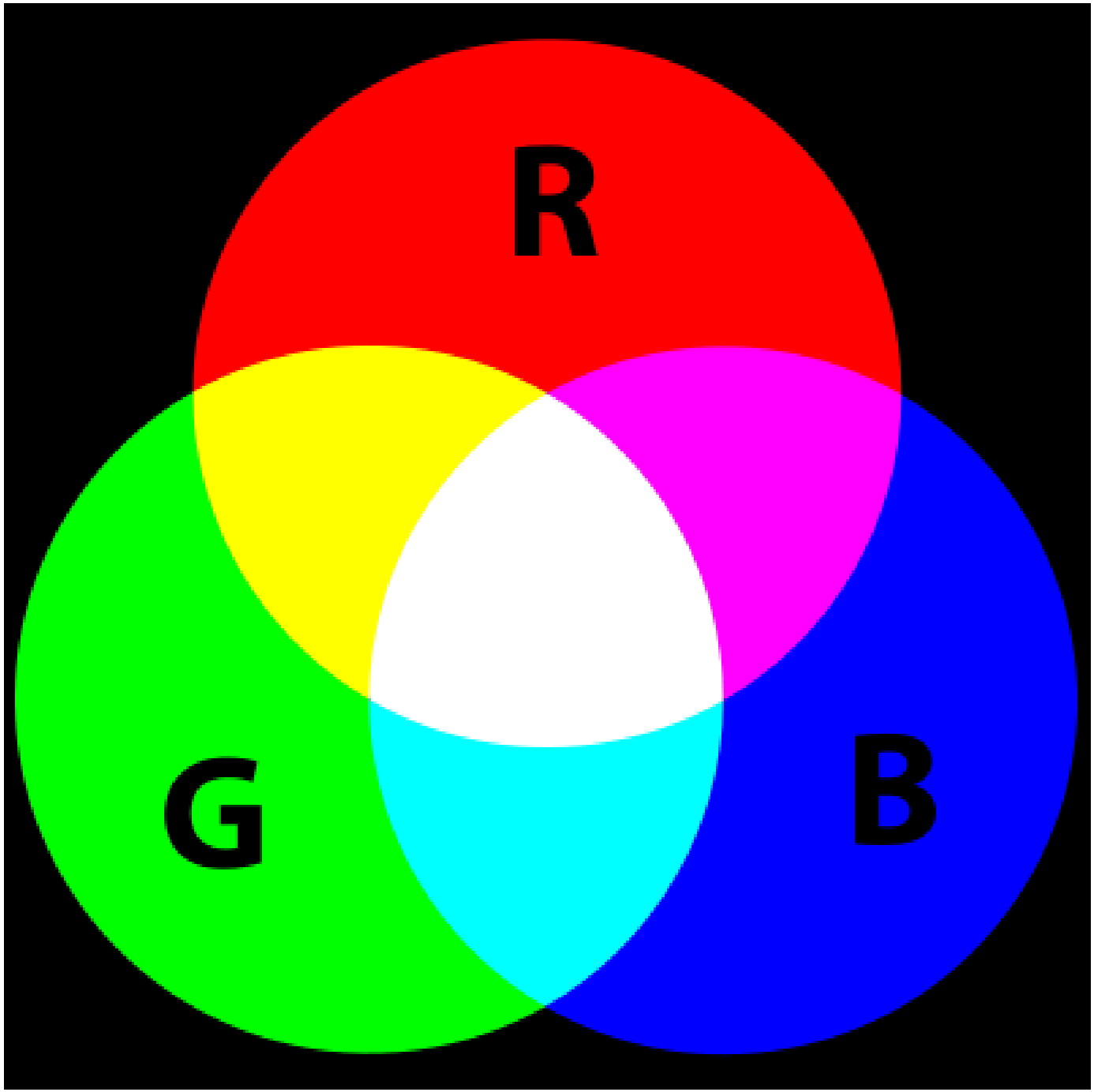
RGB* stands for Red, Green, Blue.

Each pixel in RGB has three values – a Red value, a Green value and a Blue value.

When combined, these make up a unique colour.

A 24-bit image has 8 bits per channel which makes 16.7 million colours!

* Note : in OpenCV, the order is BGR. Why is a mystery….


R
G
B


R


G


B

Colour Spaces

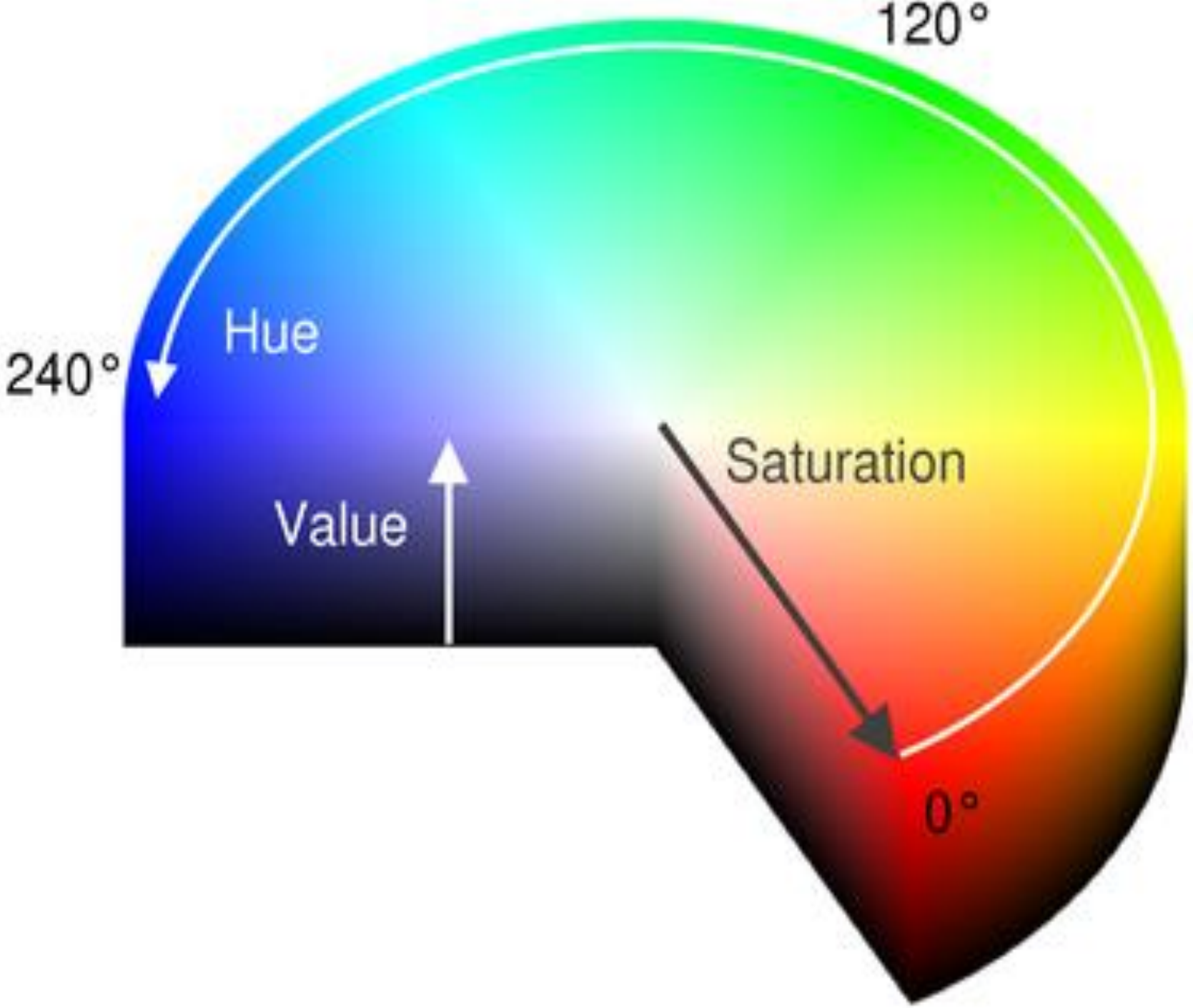
HSV

HSV (or HSI) stands for Hue, Saturation and Value (or Intensity).

Hue : describes the colour of a pixel on a colour wheel, ranging from red through green to blue

Saturation : describes the depth of the colour – is it rich or faded?

Intensity / Value : describes the brightness of the pixel


120°
240°
Hue
Value
Saturation
0°


H


S


V

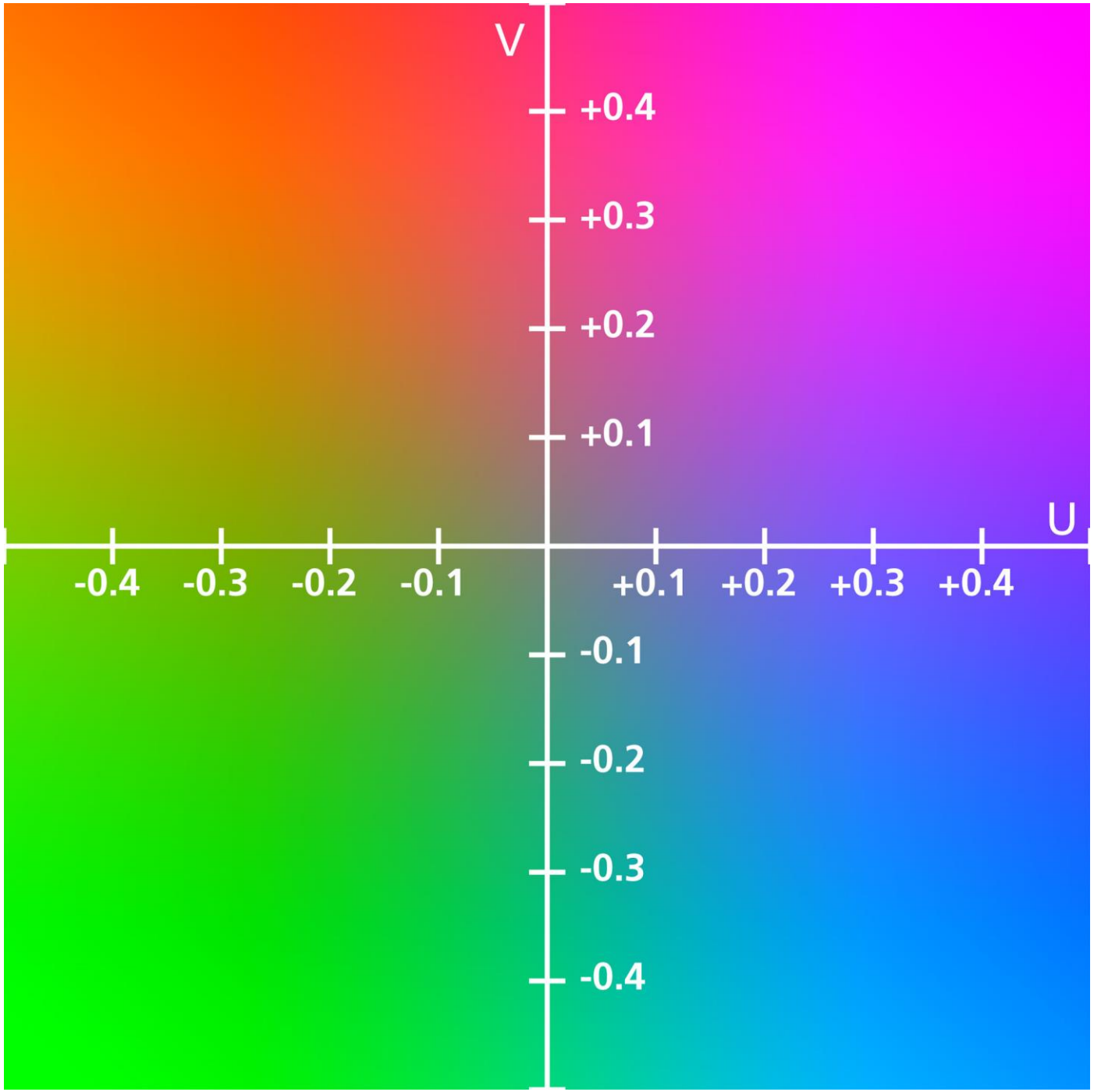
Colour Spaces

YUV

YUV contains information about the luminance (Y) and the chrominance (UV) of an image.

This can be useful in improving the apparent quality of an image as humans see luminance variations more clearly than chrominance.

It can also work well for distinguishing particular colour ranges, for example, in flesh detection.


V
+0.4
+0.3
+0.2
+0.1
U
-0.4
-0.3
-0.2
-0.1
+0.1
+0.2
+0.3
+0.4
-0.1
-0.2
-0.3
-0.4


Y


U


V

Colour Spaces

Grayscale

Grayscale images are often referred to as ‘black & white’ by the uninitiated.

Unlike the other colour spaces, a grayscale image has only one value per pixel, rather than three.

Often, grayscale represents the intensity of the image but it can represent any pixel value.

Colour Spaces

Binary

Binary images are actually ‘black & white’ since their pixels can have only two values – black or white.

Like a grayscale image, a binary image has only one value per pixel.

Usually, binary images are created using some type of segmentation.

Colour Spaces

Other Colour Spaces

There are other colour spaces besides these such as…

- YMCK (Yellow, Magenta, Cyan, Black) – used in printing
- Lab and YCbCr – both variations on YUV

In Python with OpenCV

Getting Started

To set up the environment for image processing, we first need to import two handy libraries…

OpenCV : This Open Computer Vision library is the magic sauce of this module:

```
import cv2
```

NumPy : Numerical Python helps handle the images as numbers:

```
import numpy as np
```

Getting Started

There are a few other handy libraries we will use along the way and you will probably find some nice once yourselves. For starters…

MatPlotLib : MATLAB Plotting library is a must for all MATLAB fans:

```
from matplotlib import pyplot as plt
from matplotlib import image as image
```

EasyGUI : Makes Graphical User Interfaces easy:

```
import easygui
```

Read & Write Images

Read & Write Images

Input Images

Images can be acquired for processing in three handy ways…

- Direct input
- User input
- Image capture

Read & Write Images

Direct Input

Direct input is the quickest option when you know the filename…

```
I = cv2.imread('image.jpg')
```

`I` is the image and `'image.jpg'` is the filename.

Note : quotation marks (“) can also be used around the filename

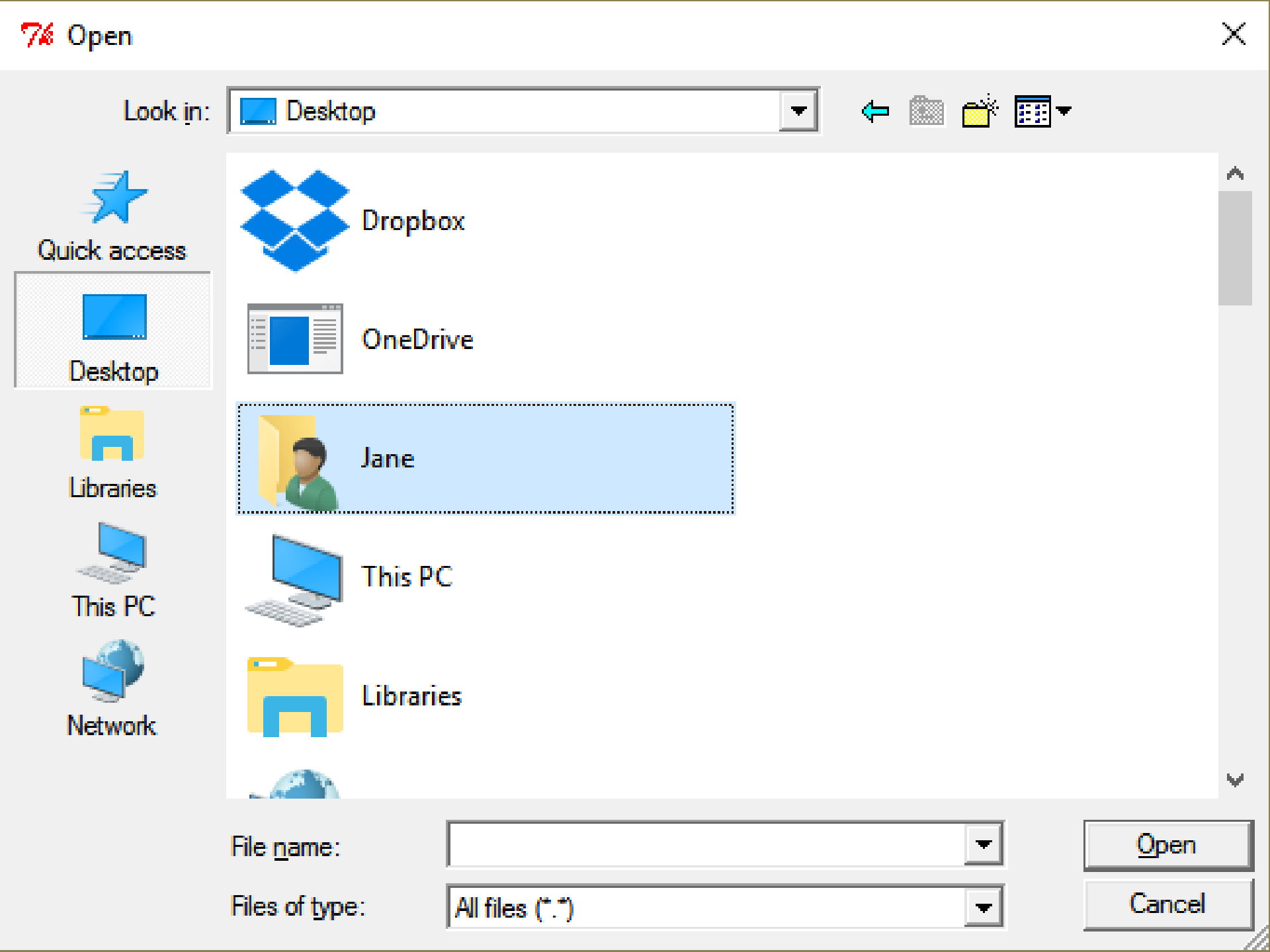
Read & Write Images

User Input

User input allows the user to select a file. This is made easy using the EasyGUI module…

```
f = easygui.fileopenbox()
I = cv2.imread(f)
```

`f` is the filename acquired by user selection.


Open
Look in:
Desktop
Quick access
Desktop
Libraries
This PC
Network
Dropbox
OneDrive
Jane
This PC
Libraries
File name:
Files of type:
All files (*.*)
Open
Cancel

Read & Write Images

Image Capture

Image capture allows images to be grabbed directly from a webcam…

```
camera = cv2.VideoCapture(0)
(grabbed, I) = camera.read()
```

`camera` is the camera object, `grabbed` is a variable used to check whether the frame was captured correctly.

Read & Write Images

Video Capture

This capture can be expanded to grab live video from a webcam or frames from a video file. This is just the same code but in a **while** loop.

```
camera = cv2.VideoCapture(0)
grabbed = True
while grabbed:
        (grabbed, I) = camera.read()
        # process or show I here
camera.release()
```

Read & Write Images

Write Images

Writing images is straightforward in OpenCV…

```
cv2.imwrite('image.jpg', I)
```

Showing Images

Showing Images

Showing images on the screen can be achieved using OpenCV or MatPlotLib…

OpenCV : `cv2.imshow("image", I)`

MatPlotLib : `plt.imshow(I)`

Showing Images

OpenCV

OpenCV's *imshow* is compatible with OpenCV's bizarre BGR order.

However, it requires a *waitKey* function to be called to stay on the screen.

This *waitKey* function waits for you to press a key before proceeding:

```
cv2.imshow("image", I)
key = cv2.waitKey(0)
```

Showing Images

MatPlotLib

MatPlotLib offers many advantages in plotting and drawing.

However, it shows images as RGB (proper order!).

This requires images to be converted from BGR to RGB.

It also requires a *show* function to be called at the end of the code:

```
I = cv2.cvtColor(I, cv2.COLOR_BGR2RGB)
plt.imshow(I)
plt.show()
```

Showing Images

Switching Colorspaces

This *cvtColor* (convert colour) function is handy for switching colour spaces from BGR…

```
RGB = cv2.cvtColor(I, cv2.COLOR_BGR2RGB)
HSV = cv2.cvtColor(I, cv2.COLOR_BGR2HSV)
YUV = cv2.cvtColor(I, cv2.COLOR_BGR2YUV)
 G = cv2.cvtColor(I, cv2.COLOR_BGR2GRAY)
```

Pixels

Pixels

Most processing we have seen so far is happening to the whole image.

However, it can be useful to be able to access or change specific pixels or a set of pixels.

The general format for pixel access is:

```
I[i,j,c]
```

`i` and `j` are the row and column locations* and `c` is the colour channel.

*Note : i is equivalent to y while j is equivalent to x

Pixels

To *access* a pixel's value, we can either look at each colour channel separately or at all three as a *tuple*…

```
B = I[i,j,0]
BGR = I[i,j]
```

`B` is the blue value at location (`i`,`j`).

`BGR` is a tuple containing the red, green and blue values

i.e. BGR = (blue , green, red).

Pixels

To *assign* a pixel's value, we can do the reverse…

```
I[i,j,0] = 255
I[i,j] = (255,0,0)
```

The first sets the blue value at location (`i,j`) to 255.

The second sets this pixel location to blue.

Pixels

The colon operator (`:`) allows a range of values to be selected.

It can also be used on its own to mean *"all values"*…

```
I[100:200,300:400] = (255,0,0)
I[i,j,:] = 255
```

The first fills all the pixels in the rectangle from (300, 100) to (400, 200) in blue.

The second sets the pixel at location (`i,j`) to white.

Task : Access Pixels

1. Read in any image;
2. Convert this image to YUV;
3. Extract the Y channel (luminance);
4. Convert the original image to grayscale (intensity);
5. Show the two on the screen using OpenCV's *imshow* function;
6. Show the luminance on the screen side by side with the intensity using MatPlotLib's *subplot* function (look this up)*.

*Note: In MatPlotLib, you will need to set the colormap to grey to deal with grayscale images: (`plt.imshow(I, cmap='gray')`).

Drawing on Images

Drawing on Images

Keep a Copy

Before you start doodling on your image, you may want to keep a backup copy to revert if things get messy.

To do this, call the *copy* function:

```
Original = I.copy()
```

`Original` is now a copy of the image `I`.

Drawing on Images

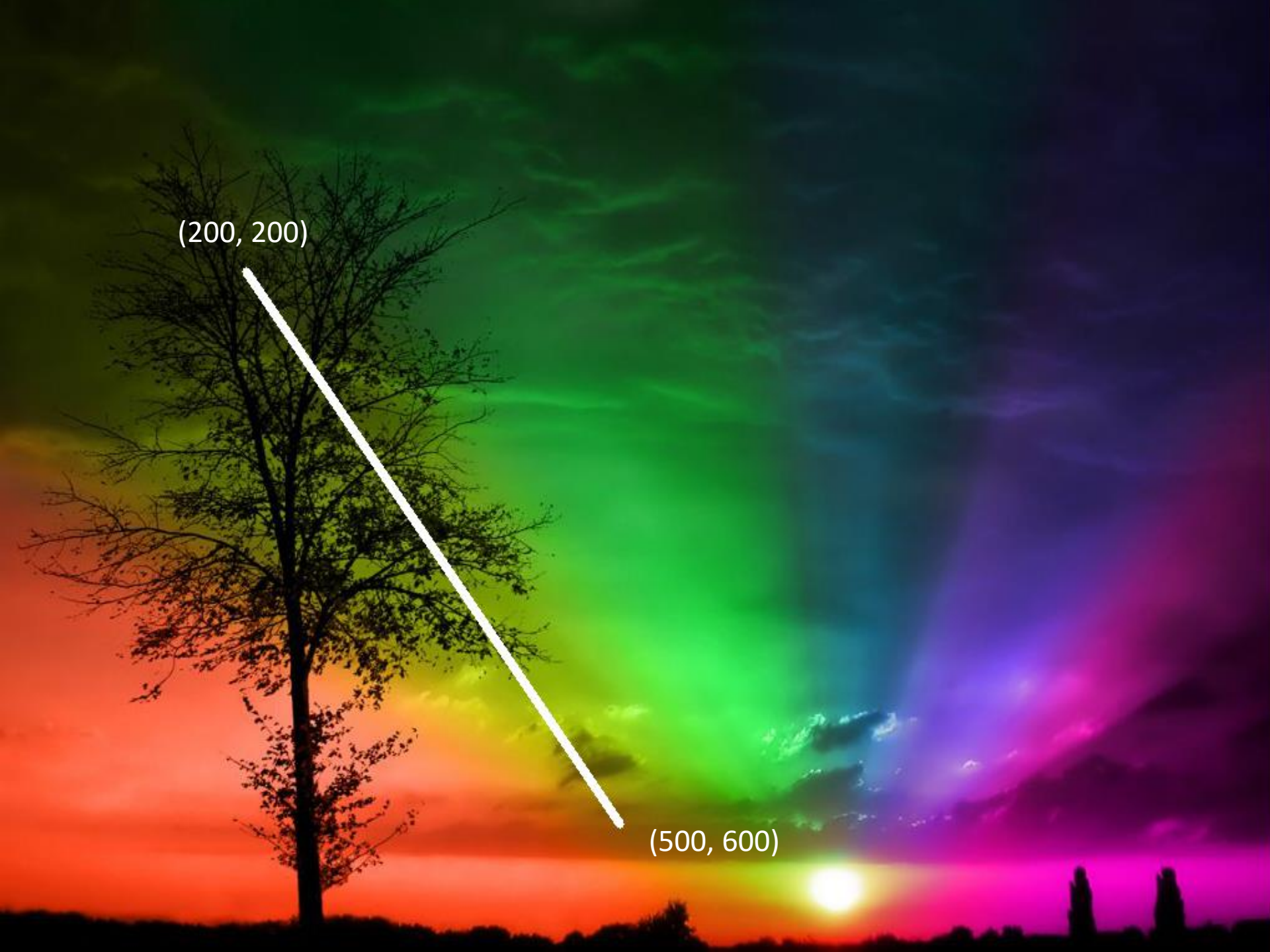
Drawing a Line

To draw a line, we need the endpoints, colour and thickness…

```
cv2.line(img = I, pt1 = (200,200), pt2 = (500,600), color = (255,255,255), thickness = 5)
```

This will draw a 5-pixel thick white line from (200,200) to (500,600).*

* Note : images start counting from the top left as they are matrices and these are (x,y) coordinates.


(200, 200)
(500, 600)

Drawing on Images

Drawing a Circle

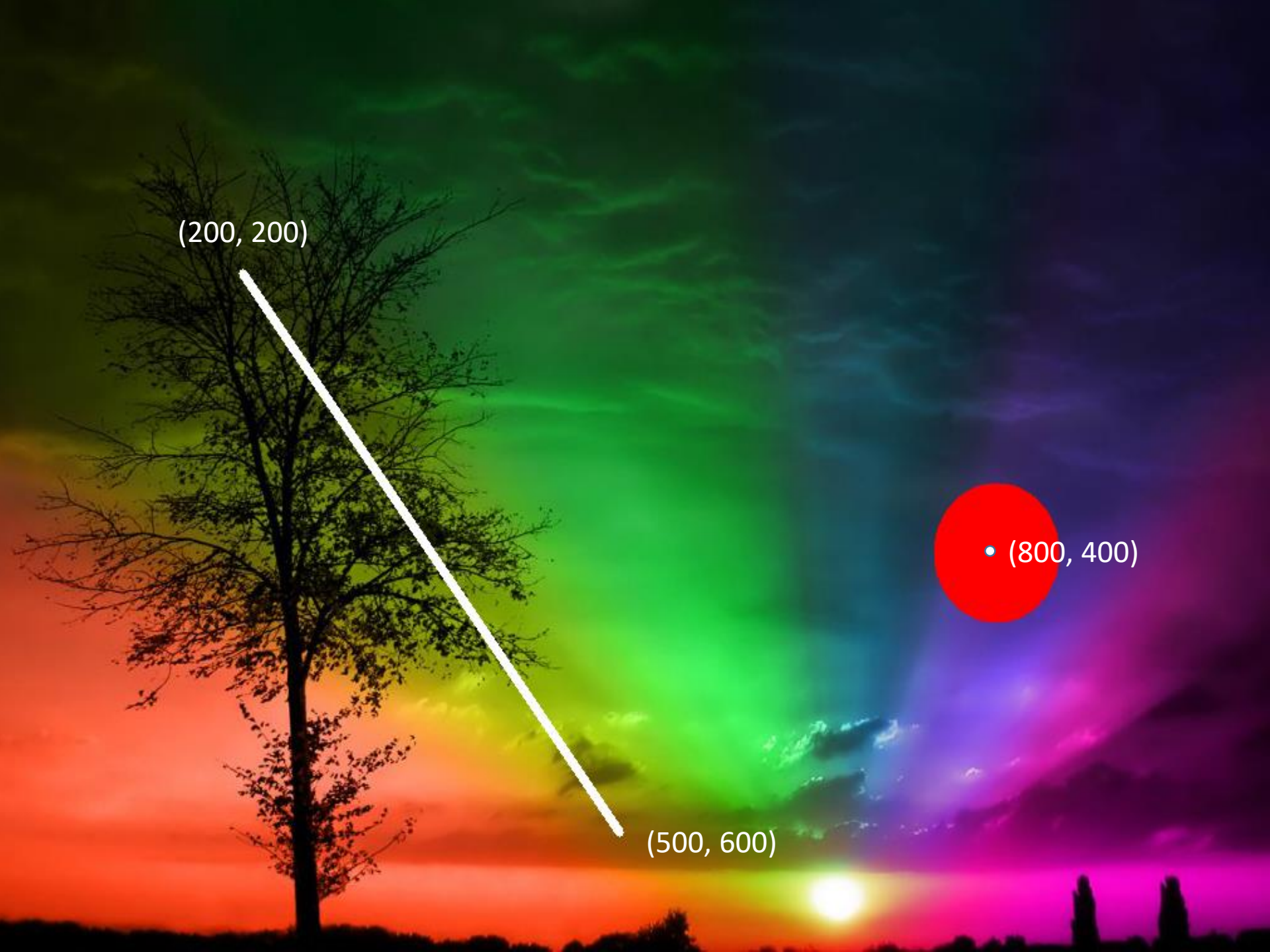
To draw a circle, we need the centre, radius, colour and thickness…

```
cv2.circle(img = I, center = (800,400), radius =
    50, color = (0,0,255), thickness = -1)
```

This will draw a red circle at (800,400) with radius 50.

A thickness of -1 leads to a filled circle.

Otherwise, thickness behaves as before in the line function.


(200, 200)
(800, 400)
(500, 600)

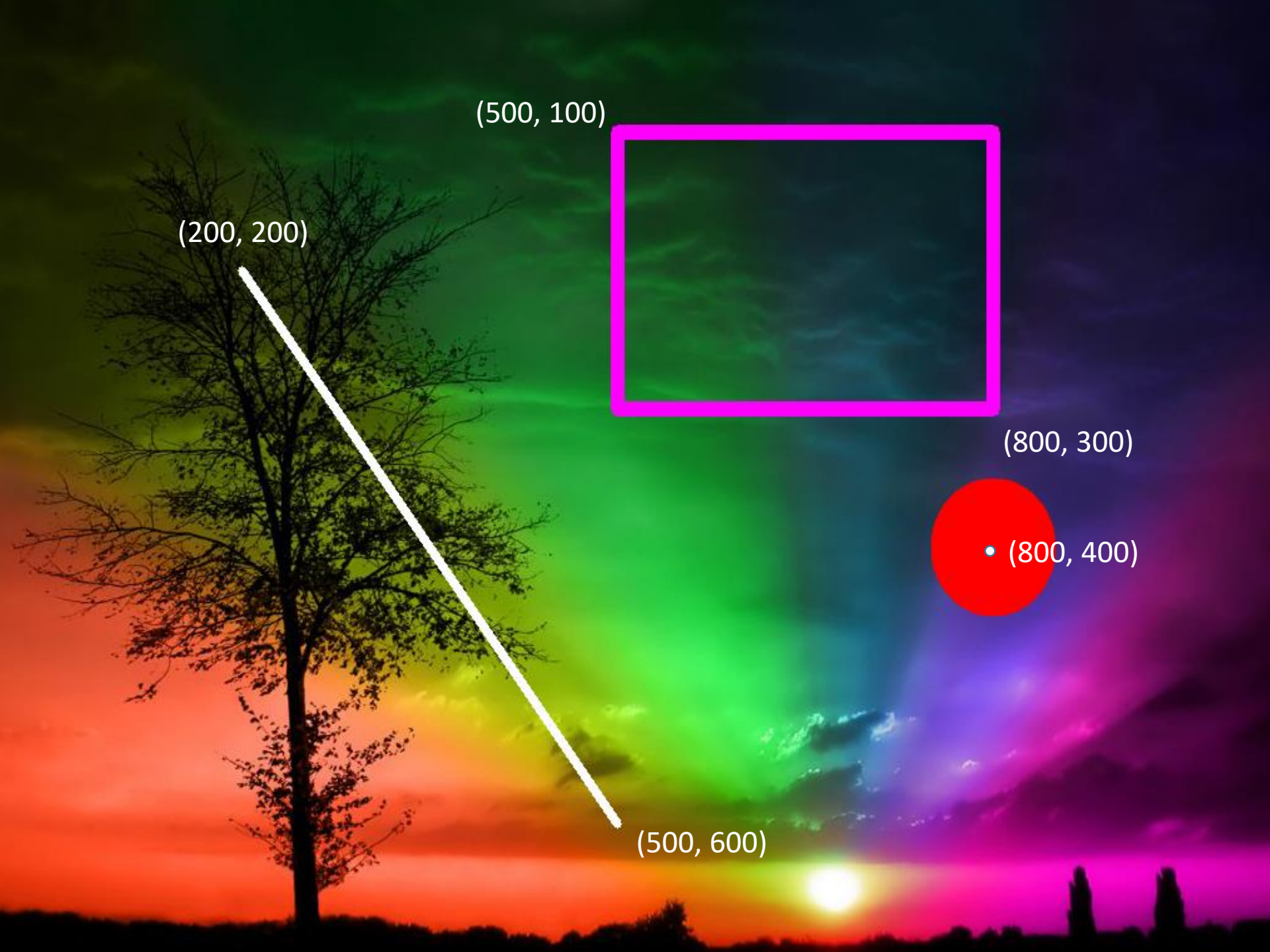
Drawing on Images

Drawing a Rectangle

To draw a rectangle, we need the top-left and bottom-right corners, colour and thickness…

```
cv2.rectangle(img = I, pt1 = (500,100), pt2 = (800,300), color = (255,0,255), thickness = 10)
```

This will draw a 10-pixel thick, magenta rectangle from (500,100) to (800,300).


(500, 100)
(200, 200)
(800, 300)
(800, 400)
(500, 600)

Drawing on Images

There are many other drawing tools available, including text boxes, ellipses, polygons, etc.

These and more drawing and plotting functions can also be performed in MatPlotLib.

For more specific drawing, we need to know how to access the pixels.

User Input

User Input

Capturing user input can be a very useful thing in designing a user interface.

We can access the location where a user has clicked using mouse events.

To do this, we need to initialise a *window* for the image and a *callback function* for the mouse event.

This involves a bit more coding …

User Input

Function for mouse events

The Point

Reacting to left button clicks

Start here….

```
def draw(event,x,y,flags,param):
    if event == cv2.EVENT_LBUTTONDOWN:
        cv2.circle(img = I, center = (x,y),
            radius = 5, color = (255,255,255),
            thickness = -1)
        cv2.imshow("image", I)

cv2.namedWindow("image")
cv2.setMouseCallback("image", draw)
```

Initialising the window

Setting the function for mouse events

Task : Handling Images

1. Open a user-selected image;
2. Show this image on the screen;
3. Capture the user's click on the image;
4. Draw a 201 x 201, 5-pixel thick red square around this location;
5. Convert the pixels within the square to YUV.

Advanced Task:

Don't fall off the edge!

Review

In this section you have learned about :

- Digital Images
- Colour Spaces
- Reading and Writing Images
- Showing Images
- Pixels
- Drawing on Images
- User Input

These topics were **implemented** and **tested** in Python with OpenCV.