📦 Importing Libraries

Before we start, we need to load some essential libraries for the project:

- **Matplotlib (pyplot, gridspec):** used to visualize images and organize plots in different layouts.
- **NumPy:** allows us to manipulate arrays and numerical data, which are the foundation for working with images in computer vision.
- **TensorFlow:** the machine learning framework that provides the tools to run neural networks and process data.
- **TensorFlow Hub:** a repository of pre-trained models. We will use it to load a *style transfer* model without training from scratch.

```
import matplotlib.pyplot as plt
from matplotlib import gridspec
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
```

🖼️ Defining the Image Paths

At this step, we specify the **files that will be used** in the *style transfer* process:

- `content_image_path` → represents the **content image** (the base that will receive the style).
- `style_image_path` → represents the **style image** (the artwork whose aesthetics will be applied).

In this example, we selected a photo named **baina2.jpeg** as the content and the painting **The Scream** as the style.
These files must be previously uploaded to the Colab environment under the `/content/` folder.

```
content_image_path = '/content/baina2.jpeg'
style_image_path = '/content/grito.jpg'
```

📥 Function to Load Images

Here we define the function `load_image`, which is responsible for **reading and preparing images** for the model:

1. `tf.io.read_file(path)` → opens the image file from the given path.
2. `tf.io.decode_image(..., channels=3, dtype=tf.float32)` → decodes the image into RGB format, converting pixel values to **float32** (required by TensorFlow).
3. `[tf.newaxis, ...]` → adds a new dimension (batch size = 1), since deep learning models expect **batches of images**.
4. `tf.image.resize(image, size, preserve_aspect_ratio=True)` → resizes the image to the desired size (default 256x256), while keeping the original aspect ratio.

🔑 This function ensures that any loaded image is in the **correct format** to be used by the *style transfer* model.

```
def load_image(path, size = (256, 256)):
  image = tf.io.decode_image(tf.io.read_file(path), channels = 3, dtype = tf.float32)[tf.newaxis, ...]
  image = tf.image.resize(image, size, preserve_aspect_ratio = True)
  return image
```

🖼️ Loading the Content and Style Images

Here we use the `load_image` function (created in the previous block) to open and prepare the images:

- `content_image` → loads the **content image** from `content_image_path`, resizing it to **384x384 pixels**. This larger size helps preserve more visual details of the base image.
- `style_image` → loads the **style image** from `style_image_path`. Since we did not provide a size parameter, it uses the function's default (`256x256`), which is enough to capture artistic patterns without compromising performance.

🔑 With this step, both images are ready to be processed by the *style transfer* model.

```
content_image = load_image(content_image_path,(384, 384))
style_image = load_image(style_image_path)
```

🔍 Checking the Image Dimensions

This command displays the **shapes** of the tensors that represent the images:

- `content_image.shape` → shows the dimensions of the content image.
- `style_image.shape` → shows the dimensions of the style image.

The output format looks like `(1, height, width, 3)`, where:

- `1` → means we only have **one image in the batch**.
- `height, width` → represent the image size after resizing.
- `3` → corresponds to the **three color channels (RGB)**.

🔑 This step ensures that both images were loaded correctly and are in the expected format for the model.

```
content_image.shape, style_image.shape
```

```
(TensorShape([1, 384, 288, 3]), TensorShape([1, 242, 256, 3]))
```

🖼️ Function to Display Images Side by Side

Here we define the `show_images` function, which organizes and displays images neatly:

1. `n = len(images)` → counts how many images will be displayed.
2. `fig = plt.figure(...)` → sets the size of the main figure.
3. `gs = gridspec.GridSpec(1, n, ...)` → creates a grid with 1 row and **n columns**, placing all images side by side.
4. Loop `for i in range(n)` → iterates over the images:
    - `ax.imshow(np.squeeze(images[i]))` → displays the image, removing extra dimensions.
    - `ax.set_xticks([]), ax.set_yticks([])` → removes axis ticks for a cleaner look.
    - `ax.set_title(titles[i])` → if titles are provided, displays them above each image.
5. `plt.show()` → renders the figure on the screen.

🔑 This function will be used multiple times to compare the content image, the style image, and the final result.

```
def show_images(images, titles = []):
  number_images = len(images)
  plt.figure(figsize = (12,12))
  gs = gridspec.GridSpec(1, number_images)
  for i in range(number_images):
    plt.subplot(gs[i])
    plt.axis('off')
    plt.imshow(images[i][0])
    plt.title(titles[i])
```

👀 Displaying the Content and Style Images

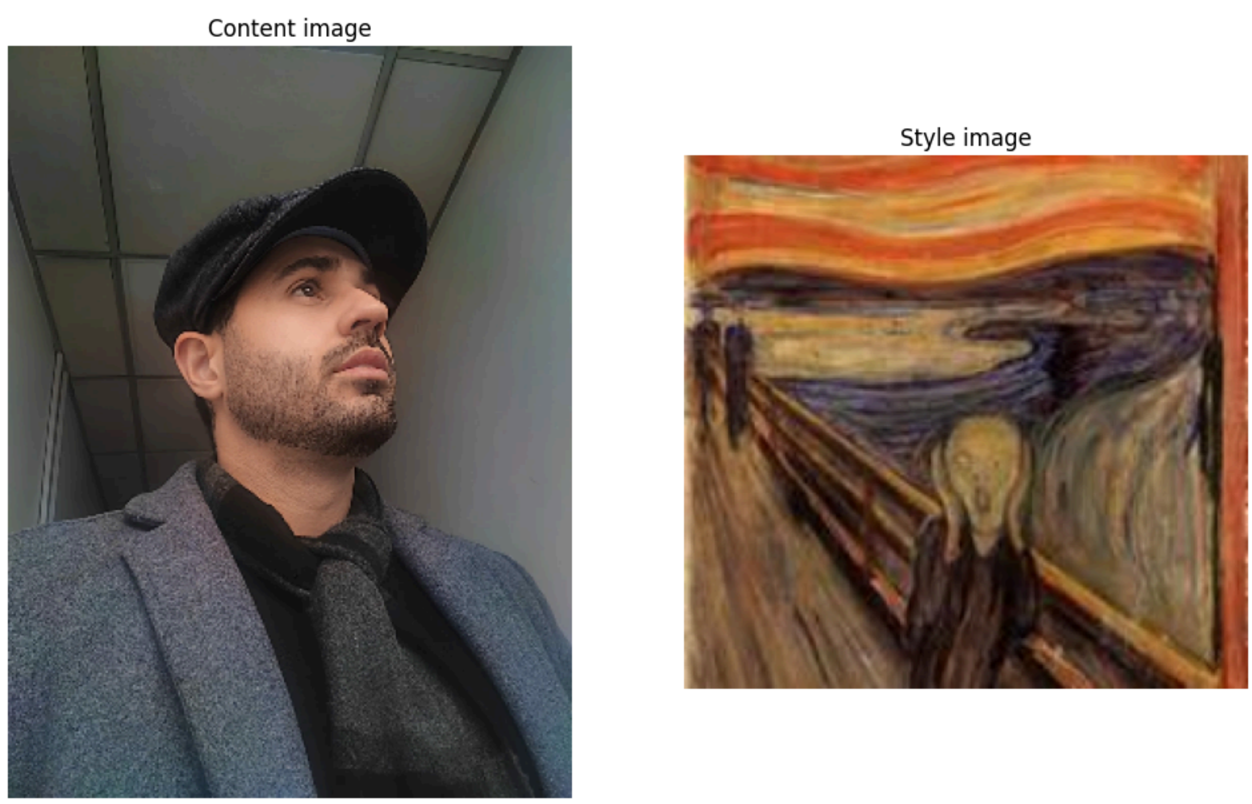
Here we call the `show_images` function to display the two main images side by side:

- `content_image` → the base image, which will provide the **content**.
- `style_image` → the **style** image, from which the artistic patterns will be extracted.

The titles `['Content image', 'Style image']` make it easy to identify each image.

🔑 This step is useful to confirm that the images were loaded correctly before applying the *style transfer*.

```
show_images([content_image, style_image], ['Content image', 'Style image'])
```



🧠 Loading the Pre-trained Style Transfer Model

In this block, we define and load the *style transfer* model from **TensorFlow Hub**:

- `model_path` → contains the link to the pre-trained **Arbitrary Image Stylization v1-256** model, developed by Google's Magenta project.
- `hub.load(model_path)` → downloads and loads the model into memory, making it ready to use.

This model has already been trained on a wide range of artistic images, allowing us to apply different styles to any content image without training from scratch.

🔑 Advantage: we save time and resources by using a robust, community-tested model.

```
model_path = 'https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2'
model = hub.load(model_path)
```

🎨 Applying the Style Transfer Model

Here we pass the loaded images into the pre-trained model:

- `tf.constant(content_image)` → converts the content image into a TensorFlow **constant tensor**.
- `tf.constant(style_image)` → does the same for the style image.
- `model(...)` → applies the *style transfer* model, producing a stylized version of the content image.

The result is stored in `results`, which is a **tensor containing the newly generated image**.

🔑 This is the core step of the project: merging **content** and **style** into a single image.

```
results = model(tf.constant(content_image), tf.constant(style_image))
```

🧐 Inspecting the Results Object

Here we simply type `results` to check what the model returned.

- The output is not the image itself, but a **TensorFlow tensor**.
- This tensor holds the numerical values (pixels normalized between 0 and 1) of the stylized image.
- Typically, `results` is displayed as something like:

```
results
```

```
[<tf.Tensor: shape=(1, 384, 288, 3), dtype=float32, numpy=
 array([[[[0.88455325, 0.690301  , 0.5053787 ],
          [0.87862027, 0.6721555 , 0.49283764],
          [0.90481526, 0.7134768 , 0.5491623 ],
          ...,
          [0.5812541 , 0.32032126, 0.16099231],
          [0.730587  , 0.48698083, 0.25199202],
          [0.75477576, 0.49209446, 0.26908016]],

         [[0.87687737, 0.66416377, 0.48129493],
          [0.86801785, 0.6467988 , 0.46174735],
          [0.9020329 , 0.7020096 , 0.52940065],
          ...,
          [0.58642364, 0.31620863, 0.15983275],
          [0.7235429 , 0.48277578, 0.24555   ],
          [0.749077  , 0.49474633, 0.26055956]],

         [[0.849659  , 0.6163755 , 0.42541593],
          [0.84391636, 0.600268  , 0.40830228],
          [0.89221114, 0.68010783, 0.49052972],
          ...,
          [0.52604455, 0.2572981 , 0.12173425],
          [0.62547237, 0.37159193, 0.1658686 ],
          [0.6699468 , 0.39346033, 0.18269347]],

         ...,

         [[0.77760214, 0.6056837 , 0.4101181 ],
          [0.8307236 , 0.7046526 , 0.49631006],
          [0.80186266, 0.6768198 , 0.42045048],
          ...,
          [0.595581  , 0.44921616, 0.411162  ],
          [0.73149025, 0.62386715, 0.5479021 ],
          [0.7128068 , 0.584705  , 0.48186162]],

         [[0.7789683 , 0.58773106, 0.40908805],
          [0.8305505 , 0.686993  , 0.4886381 ],
          [0.8094063 , 0.6705128 , 0.42074057],
          ...,
          [0.5885347 , 0.45350558, 0.4060379 ],
          [0.7279805 , 0.62011325, 0.5419871 ],
          [0.7131064 , 0.5803783 , 0.47452122]],

         [[0.79533666, 0.6030836 , 0.42951494],
          [0.84304774, 0.6991737 , 0.49969643],
          [0.81675905, 0.68528044, 0.43424067],
          ...,
          [0.5961308 , 0.46610036, 0.39684847],
          [0.7329342 , 0.622434  , 0.53084874],
          [0.7161103 , 0.5782915 , 0.46839574]]]], dtype=float32)>]
```

👁️ Comparing Content, Style, and Result

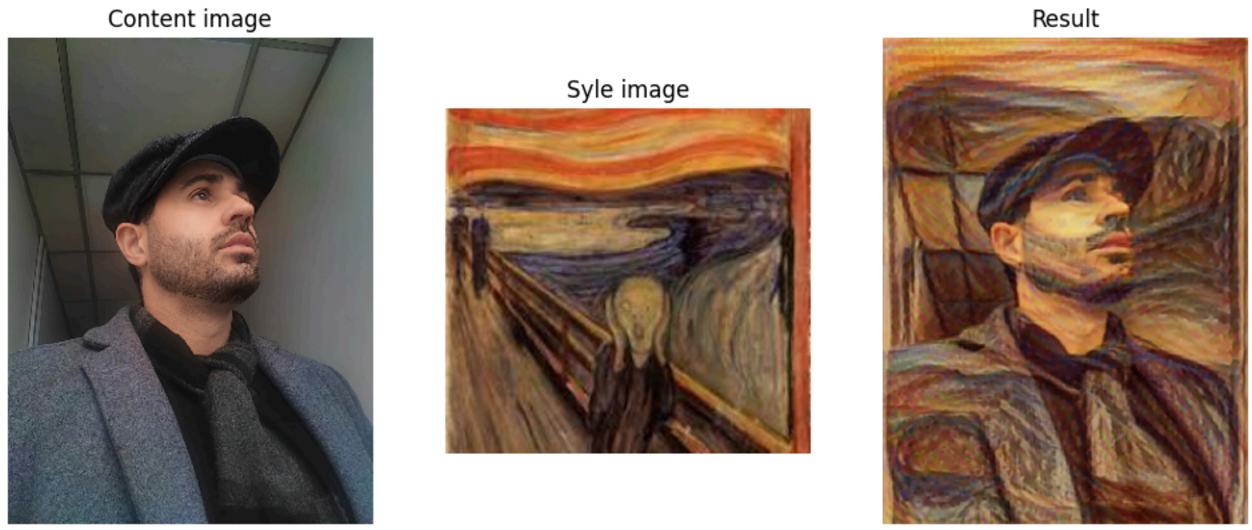
Here we call the `show_images` function again, but this time displaying **three images side by side**:

1. `content_image` → the base content image.
2. `style_image` → the chosen style image.
3. `results[0]` → the **style transfer result**, i.e., the stylized content image.

The titles `['Content image', 'Style image', 'Result']` make it easy to identify each one.

🔑 This step is crucial to clearly see how the model merged **structure (content)** and **art (style)**.

```
show_images([content_image, style_image, results[0]], ['Content image', 'Syle image', 'Result'])
```



Preparing the Result Image

Here we make a small adjustment to the tensor produced by the model:

- `results[0]` → selects the first (and only) image from the output.
- `tf.squeeze(...)` → removes the unnecessary extra dimensions, leaving the standard image format `(height, width, 3)`.

Without `squeeze`, the image shape would remain `(1, height, width, 3)`, meaning a **batch size** of 1.

🔑 This step is essential so we can save or manipulate the image as a regular file.

```
result_image = tf.squeeze(results[0])
result_image = tf.clip_by_value(result_image, 0.0, 1.0)
```

💾 Saving the Result Image

Here we save the stylized image as a PNG file:

- `result_image.numpy()` → converts the TensorFlow tensor into a **NumPy array**, which is supported by Matplotlib.
- `plt.imsave("result.png", ...)` → saves the array as an image named **`result.png`** in the current Colab directory.

Once saved, the file can be downloaded to your computer or used in other applications.

🔑 This final step turns the model output into a regular image file that can be easily shared.

```
plt.imsave("result.png", result_image.numpy())
print("Image saved as result.png")
```

```
Image saved as result.png
```

✅ Conclusion

We have completed the **Style Transfer** project using TensorFlow Hub! 🎨✨

- Loaded and prepared the **content** and **style** images.
- Applied a pre-trained model to merge structure and art.
- Visualized the results and saved the final image as a file.

This workflow demonstrates how we can leverage pre-trained models to create impressive visual effects without training a neural network from scratch.

```
print("✅ Project successfully completed! The result image has been saved as 'result.png'.")
```

```
✅ Project successfully completed! The result image has been saved as 'result.png'.
```