Learning Objectives

Learners will be able to...

- **Create complex and creative image compositions**
- **Develop advanced blending techniques with custom functions and transformations**
- **Understand the principles of image composition and blending**
- **Master image compositing with masks**
- **Apply various blending modes to create unique effects**

info

Make Sure You Know

You are familiar with Python.

Limitations

This is a gentle introduction. So there is a little bit of Python programming. The information might not be the most up to data as OpenAI releases new features.

Setting Up

We will explore various techniques to combine and blend multiple generated images, creating complex and visually engaging compositions. By the end of this lesson, you will be able to create unique and captivating images by combining the power of the DALL-E API with PIL's image manipulation capabilities.

Image composition and blending are essential techniques in digital image processing, allowing you to create new images by combining multiple images or layers. By understanding how to manipulate image layers and control their transparency and visibility, you can create a wide range of visual effects and styles. In this lesson, we will cover:

- **Image Compositing**: Learn how to combine multiple images using masks, controlling the transparency and visibility of different image layers.
- **Image Blending**: Explore various blending modes, such as multiply, screen, and overlay, to create unique effects when combining two images.
- **Advanced Blending Techniques**: Develop custom blend modes and gradient masks for greater control and flexibility in image composition.
- **Creative Image Compositions**: Apply the techniques learned in this module to design and create complex image compositions with generated images from the DALL-E 2 API.

Generate base image

We are going to import our libraries and API key then we are going to create our function.

```
import os
import openai
import secret
from PIL import Image, ImageOps,ImageChops
from io import BytesIO
import requests

openai.api_key = secret.api_key

# Generate the base image
def generate_base_image(prompt):
    response = openai.Image.create(
        prompt=prompt,
        n=1,
        size="512x512"
    )
    return response['data'][0]['url']
```

You can use the function below in your code to download images generated by the DALL-E 2 API using their URLs. The function uses the `requests` library to download the image data and `BytesIO` from the `io` module to load the data into a PIL Image object. We also save the image as a file for later use, just in case.

```
def download_image(image_url,x):
    response = requests.get(image_url)
    img_data = response.content
    img = Image.open(BytesIO(img_data))
    with open(x+'.jpg', 'wb') as handler:
      handler.write(img_data)
    return img
```

Simple Image Composition

Before diving into more advanced techniques, let's start with a simple example of image composition. In this example, we will generate two images using the DALL-E 2 API:`a moon` and `a night sky`. Then, we will overlay the moon image on top of the night sky image to create a composite image.

Generate moon and night sky images

```
moon_image_url = generate_base_image('moon')
night_sky_image_url = generate_base_image('night sky with
        stars')
```

Saving our new images

```
img_data = requests.get(moon_image_url).content
with open('moon_image.jpg', 'wb') as handler:
    handler.write(img_data)
img_data = requests.get(night_sky_image_url).content
with open('night_sky.jpg', 'wb') as handler:
    handler.write(img_data)
```

After generating your images comment out the code that call on new images to be generated, so we can keep our base images consistent.

```
#saving our images as variables.
moon_image=Image.open('moon_image.jpg')
night_sky_image=Image.open('night_sky.jpg')


# Resize the moon image to fit the composition
moon_image = moon_image.resize((200, 200), Image.ANTIALIAS)

# Ensure the moon image has the correct mode with an alpha
        channel
moon_image = moon_image.convert('RGBA')

# Overlay the moon image on top of the night sky image
night_sky_image.paste(moon_image, (150, 100), moon_image)

# Save the composed image to a file
night_sky_image.save('night_sky_with_moon.jpg')
```

Image Compositing with Masks

In this section, we will explore image compositing using masks to control the transparency and visibility of different image layers. By mastering the use of masks, you will be able to create complex and visually engaging image compositions using the DALL-E 2 API and PIL.

Alpha Masks

Alpha masks define the transparency of an image, allowing you to control which parts of the image are visible when composited with another image. In the previous example, we used the `paste()` function to overlay the moon image onto the night sky image using its own alpha channel as the mask. This ensures that only the moon's visible pixels are pasted onto the night sky image.

Custom Masks

You can also create custom masks to apply more advanced compositing techniques or control the visibility of image layers in a more precise manner. Custom masks can be created using the Image.new() function with the 'L' mode, which represents a grayscale image.

Here's an example of creating a custom mask and using it to composite two images:

```
from PIL import Image

# Generate two images using the DALL-E 2 API
image1_url = generate_base_image('forest')
image2_url = generate_base_image('sunset')

# Download and open the generated images
image1 = download_image(image1_url)
image2 = download_image(image2_url)

# Create a custom mask (grayscale gradient)
width, height = image1.size
mask = Image.new('L', (width, height))
for y in range(height):
    for x in range(width):
        mask.putpixel((x, y), x)

# Composite the two images using the custom mask
result = Image.composite(image1, image2, mask)

# Save the composited image to a file
result.save('composite_custom_mask.jpg')
```

In this example, the custom mask is a grayscale gradient from left (black) to right (white). When using Image.composite(), the black areas of the mask will take the corresponding pixels from image1, while the white areas will take the pixels from image2. Gray areas of the mask will result in a blend of the two images.

Advanced Masking Techniques

You can create more complex custom masks using a variety of techniques, such as drawing shapes, adding text, or even using another image as a mask. By combining these techniques, you can create intricate and visually stunning image compositions.

Image Blending with Different Modes

Image Blending

we will explore image blending, which involves combining two images using various blending modes to create unique visual effects and styles. This will allow you to further enhance your image compositions and unlock even more creative possibilities with the DALL-E 2 API and PIL.

Blending Modes

Blending modes determine how the colors of two images interact with each other when combined. Some common blending modes include:

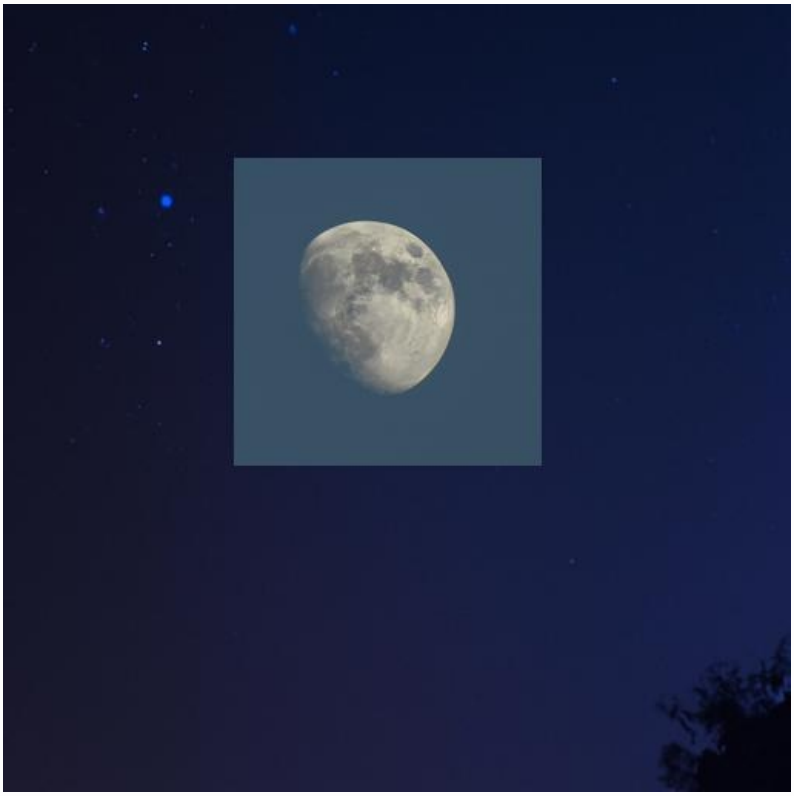
> Normal: The top image simply covers the bottom image.
> **Multiply**: The color values of the top and bottom images are multiplied, resulting in a darker image.
> **Screen**: The color values of the top and bottom images are inverted, multiplied, and then inverted again, resulting in a lighter image.
> **Overlay**: Combines Multiply and Screen modes, preserving the highlights and shadows of the bottom image.

PIL provides a built-in method called `blend()` for blending two images using the normal blending mode. However, to use other blending modes, we need to use the `ImageChops` module.

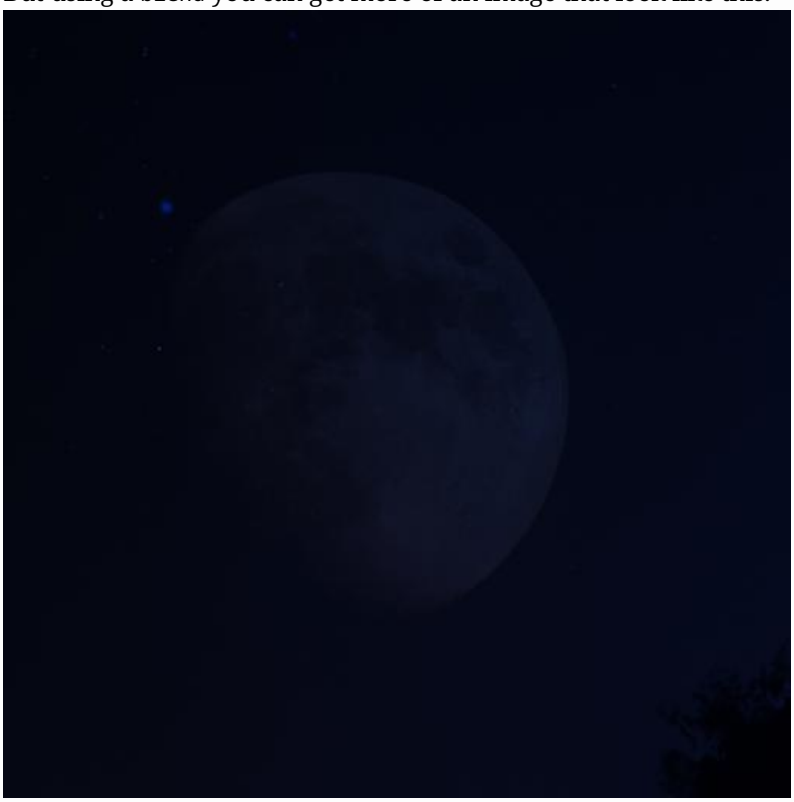
we will blend two images using various blending modes to demonstrate their effects on the final composition.

```
# Blend the images using the Multiply mode
image1 = Image.open('image1.jpg')
image2 = Image.open('image2.jpg')
multiply_blend = ImageChops.multiply(image1, image2)
multiply_blend.save('multiply_blend.jpg')
```

Your original composition pic of the moon and night sky might look like this:



But using a `blend` you can get more of an image that look like this.



```
# Blend the images using the Multiply mode
image3 = Image.open('moon_image.jpg')
image4 = Image.open('night_sky.jpg')
multiply_blend = ImageChops.multiply(image3, image4)
multiply_blend.save('multiply_blend2.jpg')
```

Custom Blend Modes

You can also create custom blend modes by defining your own functions that take the color values of the top and bottom images and return a new color value. This allows you to achieve unique and creative blending effects that may not be available with the built-in blending modes.

Here's an example of creating a custom blend mode that calculates the average of the color values of the top and bottom images:

```
from PIL import ImageChops

def average_blend(image1, image2):
    return ImageChops.add(image1, image2, scale=0.5)

# Use the custom blend mode to blend two images
image1 = Image.open('image1.jpg')
image2 = Image.open('image2.jpg')
average_blend_result = average_blend(image1, image2)
average_blend_result.save('average_blend.jpg')
```

Creating custom blend modes can be a powerful way to achieve a unique look and feel for your image compositions. Experimenting with different blending functions can lead to interesting and unexpected results, allowing you to further enhance your compositions and unlock even more creative possibilities with the DALL-E 2 API and PIL.

Advanced Blending Techniques

we will dive deeper into advanced blending techniques, such as creating custom gradient masks and blend modes, to give you greater control and flexibility in image composition.We learned how to create a custom mask for image compositing. Similarly, you can create custom gradient masks to control the blending of two images. By creating masks with varying levels of transparency, you can achieve smooth transitions between the two images.

You can create custom blend modes by defining your own functions that take the color values of the top and bottom images and return a new color value. This allows you to achieve unique and creative blending effects that may not be available with the built-in blending modes.
before we start w are going to yo create some varibale that we will use in our function.

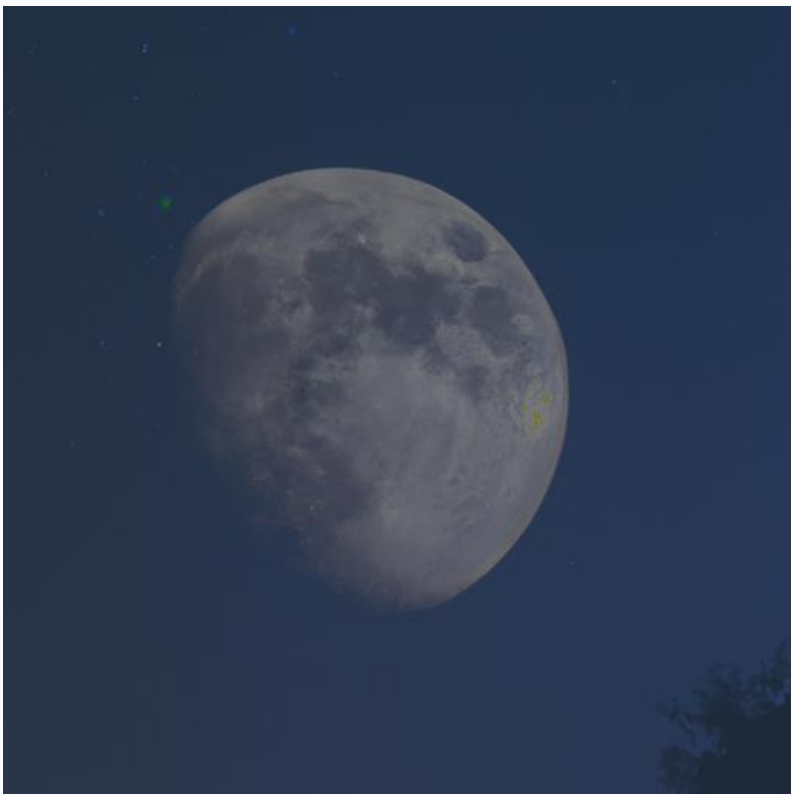
```
# Blend the images using the Multiply mode
image1 = Image.open('image1.jpg')
image2 = Image.open('image2.jpg')
# Blend the images using the Multiply mode
image3 = Image.open('moon_image.jpg')
image4 = Image.open('night_sky.jpg')
```

```
def custom_blend(image1, image2):
    return ImageChops.add_modulo(image1, image2).point(lambda x:
        x // 2)

    # Blend the images using the custom blend mode
result = custom_blend(image1, image2)
result2= custom_blend(image3, image4)
# Save the blended image to a file
result.save('custom_blend.jpg')
result2.save('custom_blend2.jpg')
```

]

This is the updated visual of the moon and night sky when using a custom blend.



Custom Blend Function

The `custom_blend` function returns the resulting image, which is the blended image with the average color value of the two input images. The `custom_blend` function takes two images, `image1` and `image2`, as inputs. Its purpose is to blend these two images together by calculating the average color value of each corresponding pixel from both images.

`ImageChops.add_modulo(image1, image2)`adds the pixel values of image1 and image2 together. It does this by adding the values of corresponding pixels in each image while ensuring that the resulting values don't exceed the maximum allowed value for each color channel. In this case, the maximum allowed value is 255 for an 8-bit image. The `add_modulo` function ensures that if the sum of two pixel values would exceed 255, the result wraps around from 0, in a similar fashion to how numbers wrap around when using the modulo operation.

The resulting image from the `add_modulo()` operation has pixel values that are double the average of the original images' pixel values. In order to get the true average, we need to divide these values by 2. We do this using the `point`() function, which applies a given function to each pixel value in the image. In this case, we use a lambda function `lambda x: x // 2` , which takes an input `x` (the pixel value) and returns the integer division of x by 2 (effectively dividing the pixel value by 2).

Combining custom gradient masks with custom blend

Combining custom gradient masks with custom blend modes opens up even more creative possibilities for your image compositions. You can use these techniques to create smooth transitions between images or to blend images in unique and visually interesting ways.

Here's an example of creating a custom gradient mask and using it with the custom_blend function we defined earlier:

```
# Create a custom gradient mask (grayscale gradient)
width, height = image1.size
mask = Image.new('L', (width, height))
for y in range(height):
    for x in range(width):
        mask.putpixel((x, y), x)

# Apply the custom gradient mask to the second image
masked_image2 = Image.composite(image2, Image.new('RGB',
        image2.size), mask)

# Blend the images using the custom blend mode
result = custom_blend(image1, masked_image2)

# Save the blended image to a file
result.save('custom_blend_gradient.jpg')
```

In this example, the custom gradient mask is applied to the second image, creating a smooth transition from left (transparent) to right (opaque). The resulting masked image is then blended with the first image using the custom_blend function. This creates a smooth transition between the two images with a unique blending effect.

Feel free to experiment with different gradient masks and blend modes to create intricate and visually stunning image compositions using the DALL-E 2 API and PIL. By combining these advanced techniques, you can unlock even more creative possibilities and take your image compositions to the next level.