# Lecture#2

May 3, 2023

## 1 Python for Signal Processing

Danilo Greco, PhD - danilo.greco@uniparthenope.it - University of Naples Parthenope

### 1.0.1 Lecture 2

This lecture will provide an overview on: 1. some basic image manipulation tasks with python 2. basic object oriented programming in python

## 2 Basic image manipulation

Many libraries are available in python to perform image manipulation: we'll be discussing about some of the most frequently used, bearing in mind that they're not the only ones.

### 2.0.1 matplotlib

Matplotlib is the standard plotting library for creating static, animated, and interactive visualizations in Python. Usually, it gets installed along with the basic scientific libraries and therefore is generally available and does not require installation. Nevertheless it can be installed, as explained above, by the following command:

pip install matplotlib

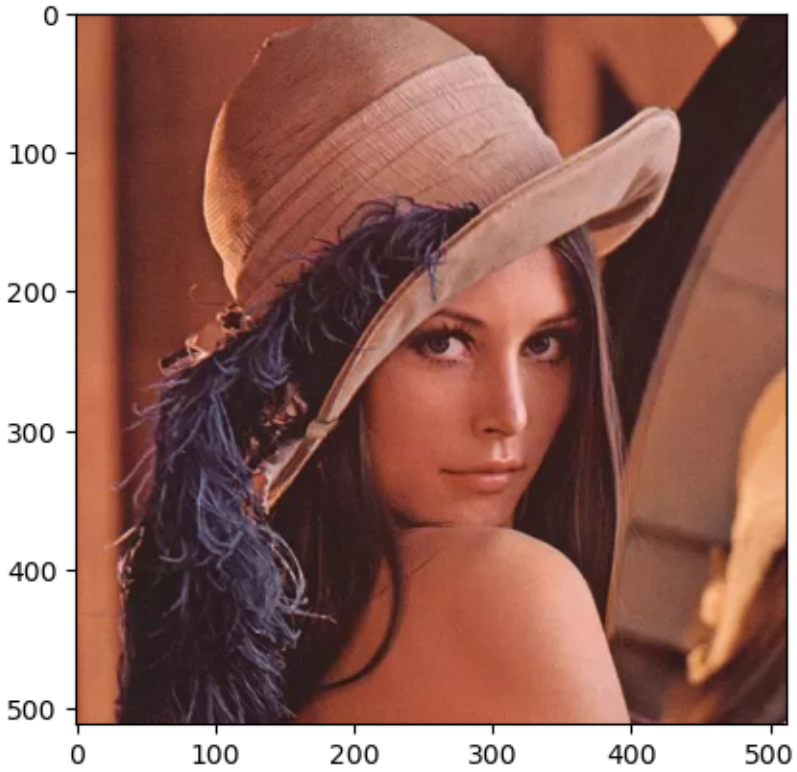
In order to turn on inline plotting, a specific magic command should be used and some imports are required:

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.image as mpi
```

**Loading images** Images can be easily loaded from a disk file with the following python code:

[4]:
```
img = mpi.imread('lena.jpg')
plt.axis("on") # removes the axis and the ticks
plt.imshow(img)
```

[4]:
```
<matplotlib.image.AxesImage at 0x1a5d122ae20>
```




The image is now represented by a 3D array whose dimensions can be displayed by printing its shape property, representing the number of rows, the number of columns and the 3 color planes for R, G and B respectively:

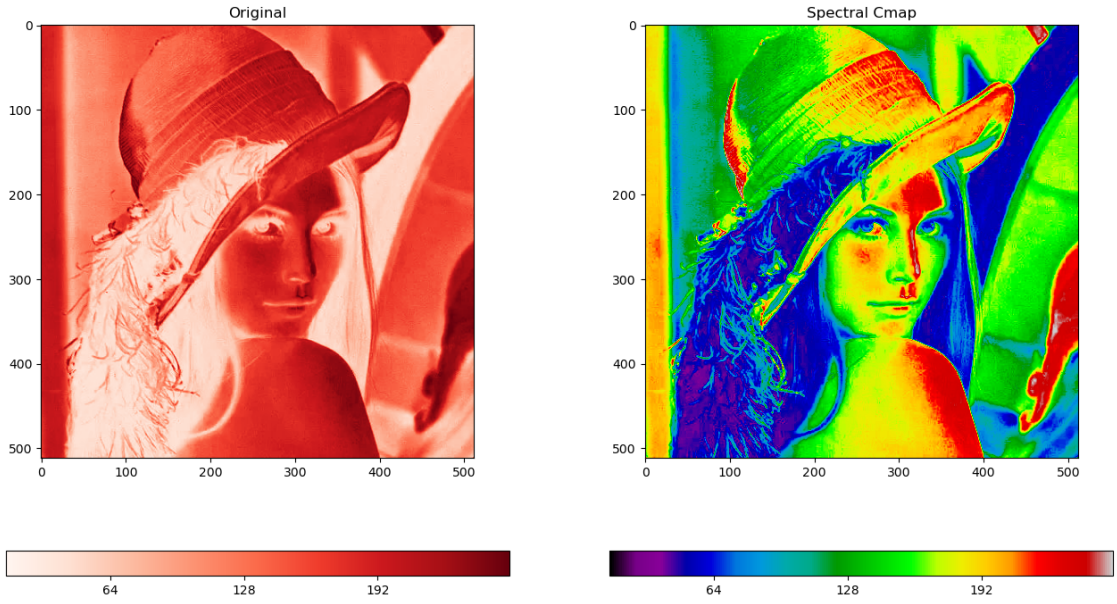
[5]:
```
print(img.shape)
```

```
(512, 512, 3)
```

We can also use pseudocolors to display each image channel and we can plot two images on the same row with subplots. Let's plot the original red channel and its version on a different color map:

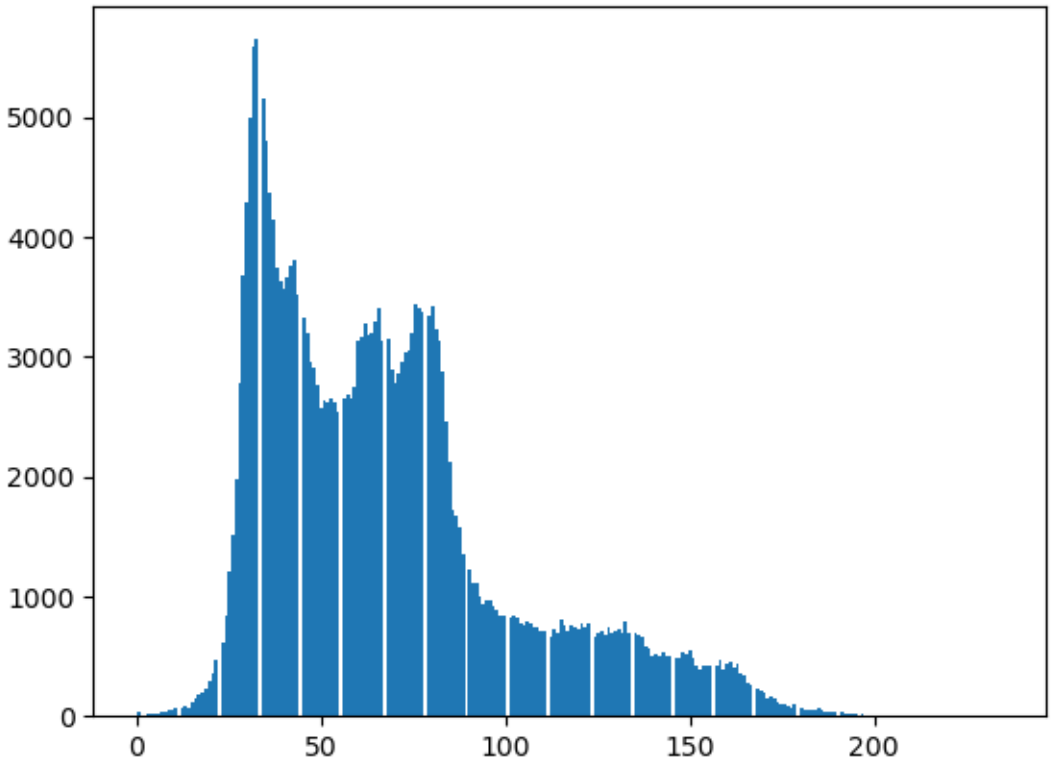
[6]:
```
rch = img[:,:,0]
fig = plt.figure(figsize=(16,9))

ax = fig.add_subplot(1, 2, 1)
imgplot = plt.imshow(rch, cmap="Reds")
ax.set_title('Original')
plt.colorbar(ticks=[0, 64, 128, 192], orientation='horizontal')

ax = fig.add_subplot(1, 2, 2)
imgplot = plt.imshow(rch, cmap="nipy_spectral")
ax.set_title('Spectral Cmap')
plt.colorbar(ticks=[0, 64, 128, 192], orientation='horizontal')
```

[6]: <matplotlib.colorbar.Colorbar at 0x1a5d12ed7c0>




And finally, it is possible to display the histogram of color distribution for our image:

[9]:
```
himg = img[:,:,2]
plt.hist(himg.ravel(),bins=256)
plt.show()
```




### 2.0.2 Numpy

Numpy is a powerful library for array management, efficient and comprehensive mathematical functions which can play an important role in image manipulation because images are de-facto arrays.

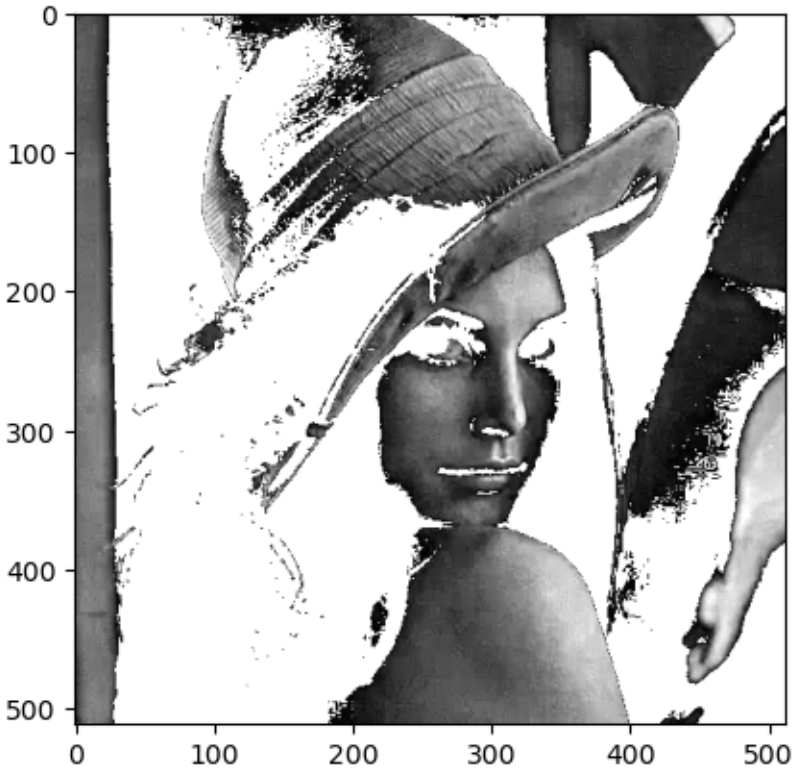
[10]:
```
print(type(img))
```

```
<class 'numpy.ndarray'>
```

Images can be loaded by matplotlib or other libraries but Numpy provides an easy way to manipulate images at pixel level. Let's say for instance that we want to apply selective filtering on color values by setting to white all pixels that contain values below a certain threshold.

[13]:
```
masked_img = rch.copy()
mask = masked_img < 150
masked_img[mask]=255
plt.imshow(masked_img, cmap='gray')
```

[13]:
```
<matplotlib.image.AxesImage at 0x1a5d16d29a0>
```



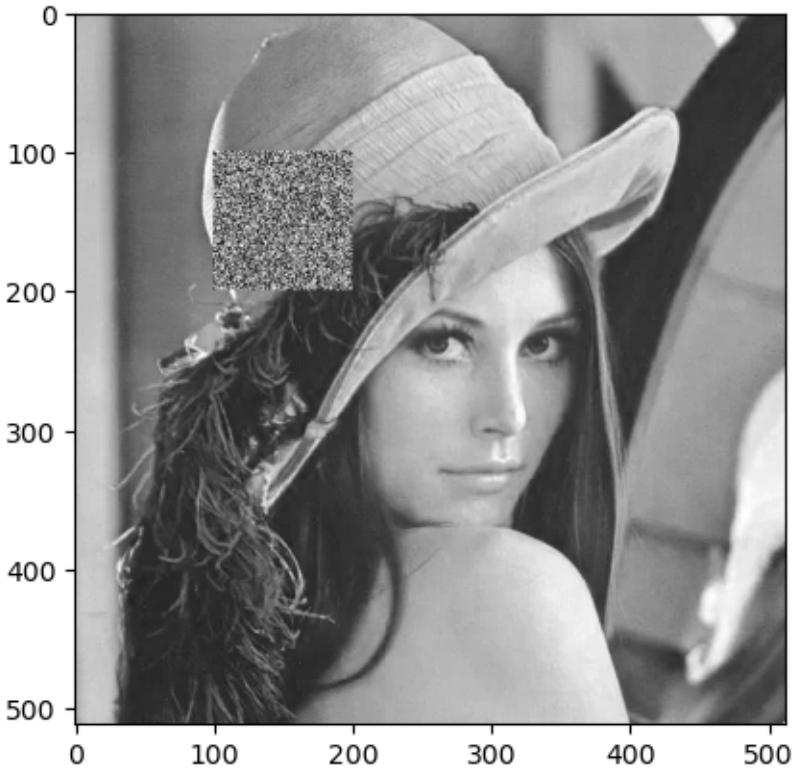

[14]:
```
mask
```

[14]:
```
array([[False, False, False, …,  True,  True, False],
       [False, False, False, …,  True,  True,  True],
       [False, False, False, …,  True,  True,  True],
       …,
       [False, False, False, …,  True,  True,  True],
       [False, False, False, …,  True,  True,  True],
       [False, False, False, …,  True,  True, False]])
```

It's also possible to replace part of our image with other image pixels. The following example uses numpy to generate random values as a replacement for the box from (200,200) to (300,300):

[16]:
```
import numpy as np

boxed_img = rch.copy()
boxed_img[100:200,100:200]=np.random.randint(0,255,size=(100,100))
plt.imshow(boxed_img, cmap='gray')
```

[16]:
```
<matplotlib.image.AxesImage at 0x1a5d1397b50>
```



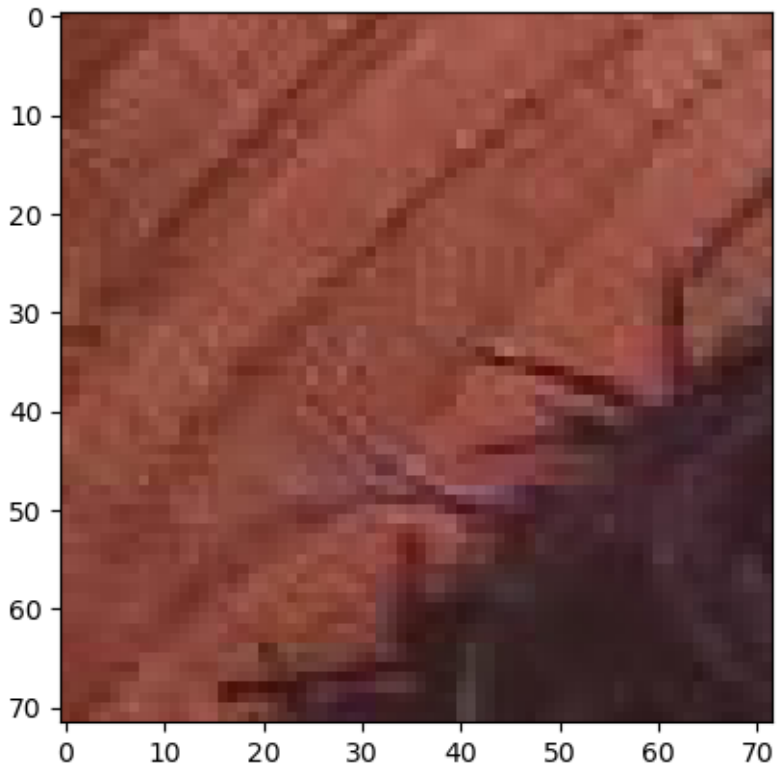

[17]:
```
boxed_img.shape
```

[17]:
```
(512, 512)
```

or to trim parts of the image:

[19]:
```
trimmed = img[128:200, 128:200]
plt.imshow(trimmed)
```

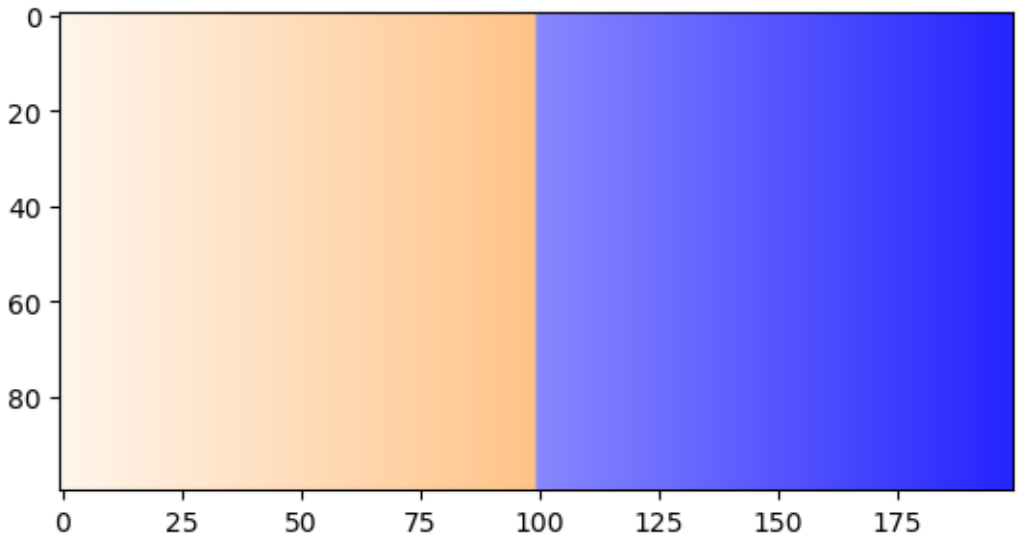
[19]:
```
<matplotlib.image.AxesImage at 0x1a5d13e5550>
```




Creating RGBA images, where A is the transparency of the image aka alpha channel, in numpy is quite easy:

[21]:
```
import numpy as np

rgba = np.zeros([100, 200, 4], dtype=np.uint8)
rgba[:,:100] = [255, 128, 0, 255] #Orange left side
rgba[:,100:] = [0, 0, 255, 255]   #Blue right side

# Set transparency depending on x position
for x in range(200): # columns
    for y in range(100): # rows
        rgba[y, x, 3] = x+20

plt.imshow(rgba)
```

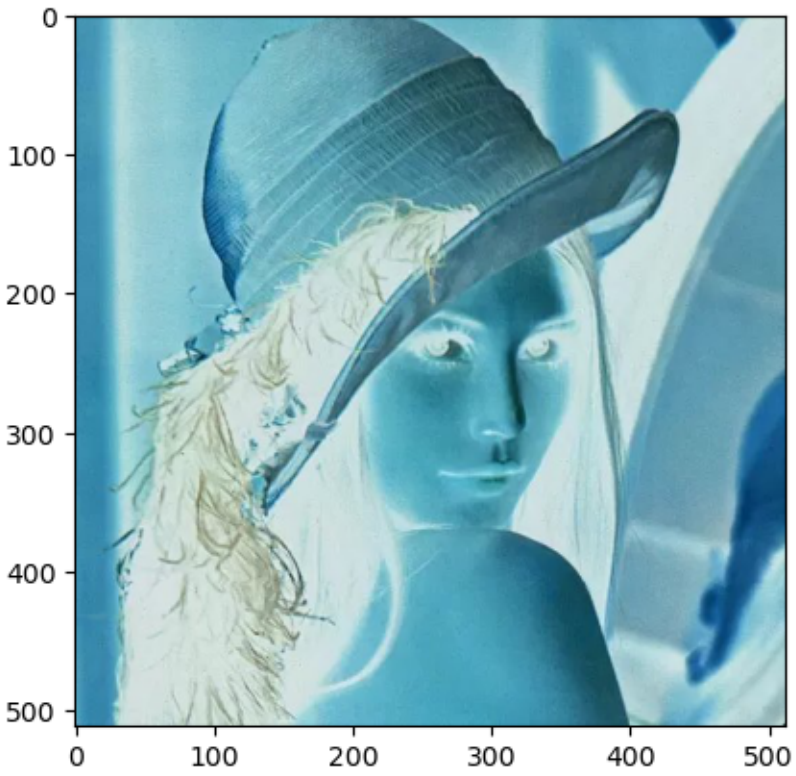
[21]:
```
<matplotlib.image.AxesImage at 0x1a5d142bca0>
```




Inverting image colors can be easily done thanks to the broadcasting capabilities of numpy. In the example all RGB planes are inverted independently:

[22]:
```
iimg = 255 - img
plt.imshow(iimg)
```

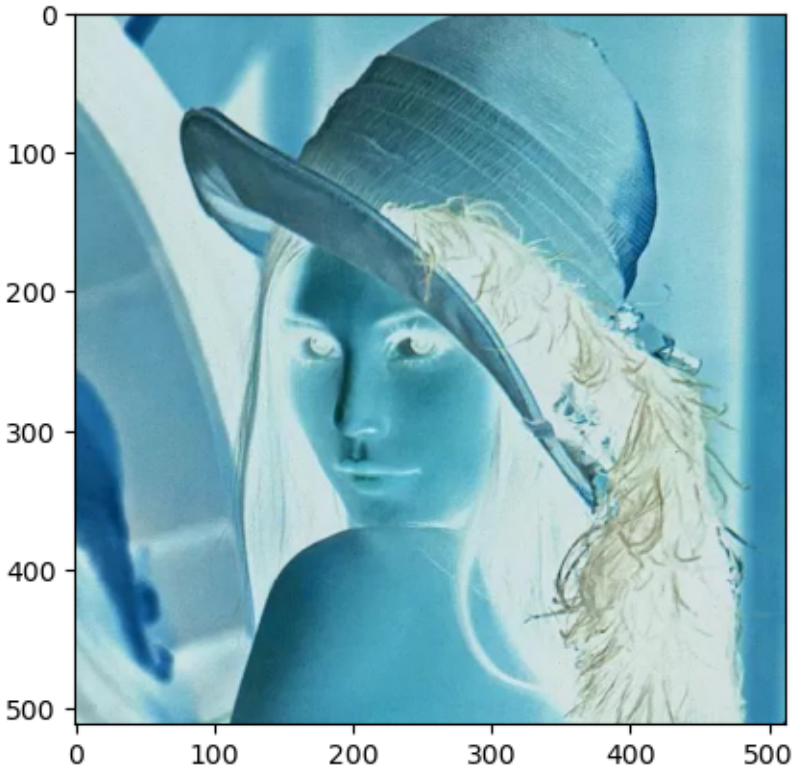
[22]:
```
<matplotlib.image.AxesImage at 0x1a5d172d880>
```




An image can also be mirrored on a specified axis with the flip function:

[24]:
```
plt.imshow(np.flip(iimg,axis=1))
```

[24]:
```
<matplotlib.image.AxesImage at 0x1a5d165a7f0>
```




The RGB values of a single pixel can be determined by assigning multiple variables from image array at desired coordinates, such as:

[26]:
```
R,G,B=iimg[100,100]
R,G,B

# in other languages I would have done
R = iimg[100,100,0]
G = iimg[100,100,1]
B = iimg[100,100,2]
```

### 2.0.3 Scipy

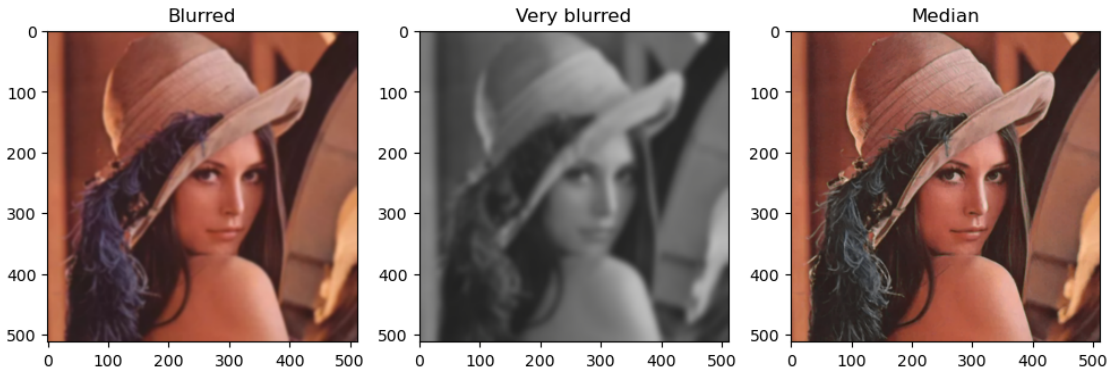
Last tool of this overview is Scipy that is a Python-based ecosystem of open-source software for mathematics, science, and engineering and can be used for basic image manipulation and processing tasks.

In particular, the submodule scipy.ndimage provides functions operating on n-dimensional NumPy arrays. The package currently includes functions for linear and non-linear filtering, binary morphology, B-spline interpolation, and object measurements.

The following example applies a gaussian filter to the image with different kernel sizes and a median filter for denoising:

[29]:
```
from scipy import ndimage
import imageio as io

lena = io.imread('lena.jpg')

blurred_face = lena.copy()
# each color plane must be filtered independently to get a color output
blurred_face[:,:,0] = ndimage.gaussian_filter(lena[:,:,0], sigma=2)
blurred_face[:,:,1] = ndimage.gaussian_filter(lena[:,:,1], sigma=2)
blurred_face[:,:,2] = ndimage.gaussian_filter(lena[:,:,2], sigma=2)

# working on the image as a whole makes it gray scale
very_blurred = ndimage.gaussian_filter(lena, sigma=5)

# apply median filter on the image
median = ndimage.median_filter(lena,3)

fig = plt.figure(figsize=(12,5))

ax = fig.add_subplot(1, 3, 1)
imgplot = plt.imshow(blurred_face)
ax.set_title('Blurred')

ax = fig.add_subplot(1, 3, 2)
imgplot = plt.imshow(very_blurred)
ax.set_title('Very blurred')

ax = fig.add_subplot(1, 3, 3)
imgplot = plt.imshow(median)
ax.set_title('Median')

plt.show()
```

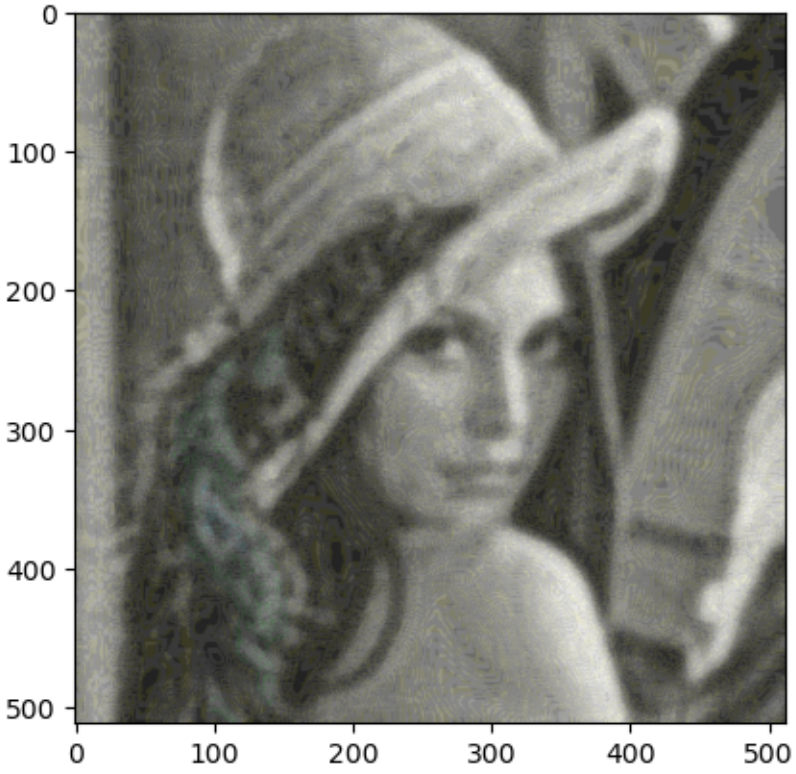
```
C:\Users\GRCDNL71D14D969B\AppData\Local\Temp\ipykernel_15776\3011725062.py:4:
DeprecationWarning: Starting with ImageIO v3 the behavior of this function will
switch to that of iio.v3.imread. To keep the current behavior (and make this
warning dissapear) use `import imageio.v2 as imageio` or call
`imageio.v2.imread` directly.
  lena = io.imread('lena.jpg')
```




And this is a possible way of sharpening the image by increasing the weight of edges by adding an approximation of the Laplacian:

[36]:
```
filter_blurred = ndimage.gaussian_filter(very_blurred, 1)
alpha = 20
sharpened = very_blurred + alpha * (very_blurred - filter_blurred)
plt.imshow(sharpened)
```

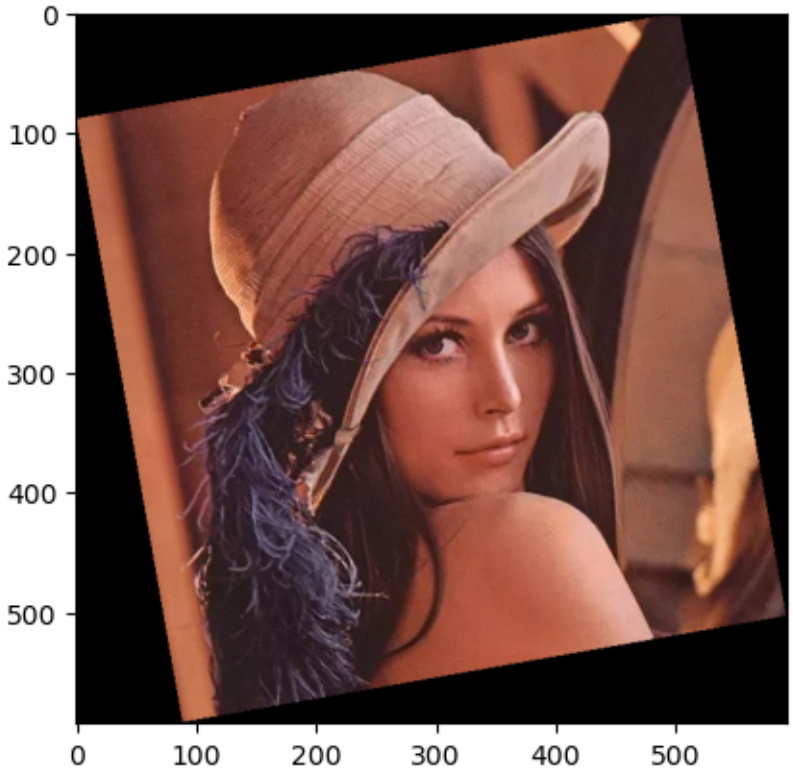
[36]:
```
<matplotlib.image.AxesImage at 0x1a5d416bf40>
```



Rotation of the image is easily done with a single function call:

[38]:
```
rotated_lena = ndimage.rotate(lena, 10)
plt.imshow(rotated_lena)
```

[38]:
```
<matplotlib.image.AxesImage at 0x1a5d41fe610>
```




## 3 Object Oriented Programming

Object-oriented programming is a paradigm that structures programs so that **properties** and **behaviors** are bundled into individual objects.

For instance, an object can represent a dog whose properties are name and breed and behaviors are bark, run, etc.

OOP models real-world entities as software objects, focusing on creating reusable code.

Objects are represented by **classes** that contain: - **properties** that represent their characteristics, usually associated to internal variables - **methods** that express the actions they can perform, that is how an object behaves, and are related to functions.

A **class** is the blueprint of an object and does not contain any valuable data: it simply describes a template that identifies object attributes and activities.

An object of a given class is called an **instance** of that class: an instance contains real data according to the given description.

> mydog = Dog("jack") and yourdog = Dog("jill") could be two different instances of the same class

A class can be defined by the class keyword and must contain the ___init___ function which is the **constructor** or the function that gets called any time a new object instance is created.

**NOTE that every function in a class definition, must have *self* as the first argument but it is never explicitly assigned when invoking the method**

[39]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        msg = 'Bau! I am {} and my age is {}'.format(self.name,self.age)
        return msg
```

The ___repr___ method (which is optional) represents the printable version of an object and can be used implicitly when printing an object, as in the following example:

[40]:
```
mydog = Dog("jack",2)
print(mydog)
```

```
Bau! I am jack and my age is 2
```

An object can execute actions through its methods; a dog can bark therefore we can add this action to the class and invoke it on the object instance:

[41]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        msg = 'Bau! I am {}'.format(self.name)
        return msg

    def bark(self):
        print('Bau! Bau! Bau!')

mydog = Dog("jack",2)
mydog.bark()
```

```
Bau! Bau! Bau!
```

The attributes defined in the ___init___ method are called **instance attributes** but it is also possible to declare **class attributes** such as species in the following example.

[42]:
```
class Dog:

    species = 'Canis familiaris'  # public attribute

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        msg = 'Bau! I am {}'.format(self.name)
        return msg

    def bark(self):
        print('Bau! Bau! Bau!')

mydog = Dog("jack",2)
mydog.species
```

[42]:
```
'Canis familiaris'
```

The difference is quite simple: **class** attributes should contain values that apply to all instances of the class whereas **instance** attributes are specific to a given instance. In the example above, *species* is an attribute of all *Dog* instances.

Regardless of their type, attributes are accessed in the same way.

[43]:
```
print('{} is of species {}'.format(mydog.name,mydog.species))
```

```
jack is of species Canis familiaris
```

### 3.0.1 Encapsulation

Encapsulation restricts access to attributes and methods and prevent data from direct modification.

It's a good practice in OOP to access attributes through **getter** and **setter** methods to preserve data **encapsulation** and control the values before assignment: usually attributes are **private** and can be retrieved by a proper getter method or modified using a setter one.

This is not the case in python where all attributes are meant to be **public**.

Even though it is possible to create private properties of a class by prepending the name with a double underscore:

[44]:
```
class Dog:

    __species = 'Canis familiaris'  # private attribute

    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

    def get_species(self):
        return self.__species

mydog = Dog("jack",2)
print(mydog.__species) # this call is going to fail
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[44], line 13
     10         return self.__species
     12 mydog = Dog("jack",2)
---> 13 print(mydog.__species)

AttributeError: 'Dog' object has no attribute '__species'
```

Accessing the property directly produces an error but the variable is properly retrieved by calling the getter method we defined:

[45]:
```
mydog.get_species() # this one works
```

[45]:
```
'Canis familiaris'
```

A better way of implementing private properties in python is by means of decorators that we can place before getter and setter definitions. These methods must be named as we want the attribute to be exposed outside the class and @property decorator precedes the getter declaration while @x.setter is placed before the setter method of the x variable.

[46]:
```
class Dog:

    __species = 'Canis familiaris'  # private attribute

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @property
    def species(self):
        return self.__species

    @species.setter
    def species(self,x):
        self.__species = x

mydog = Dog("jack",2)
```

```
print(mydog.species)

mydog.species = 'Canis Lupus'
print(mydog.species)
```

```
Canis familiaris
Canis Lupus
```

**NOTE: to invoke getter and setter methods from a child class, they must be called by prepending parent class's name.**

### 3.0.2 Inheritance

A fundamental process of OOP is **inheritance** which is a way of creating a new **derived** class that extends its **base** class.

Each class is initialized by its constructor and, when necessary, the derived class can invoke the parent class method with the super keyword.

The whatisThis method is defined both in the parent and in the child class: the child class can modify a behavior of the parent one by rewriting the content of a method.

Next, the parent class' move method can be invoked with an instance of a child class because it inherits all properties and methods of its parent.

Finally, the child class can extend the behavior of the base class by adding new methods, such as moveFaster.

[28]:
```
# parent class
class Vehicle:

    def __init__(self):
        print("init vehicle")

    def whatisThis(self):
        print("vehicle")

    def move(self):
        print("vehicle moves")

# derived class
class Car(Vehicle):

    def __init__(self):
        # call super() function
        super().__init__()
        print("init car")

    def whatisThis(self):
        #super().whatisThis()
```

```
        print("car")

    def moveFaster(self):
        print("car moves faster")

mycar = Car()
mycar.whatisThis()
mycar.move()
mycar.moveFaster()
```

```
init vehicle
init car
car
vehicle moves
car moves faster
```

### 3.0.3 Polymorphism

Another important characteristic of OOP is **polymorphism** which allows to use a common interface for multiple data types.

In the example above, the whatisThis method has been **overridden** in the child class so that it produces different outputs according to the class of each object instance.

If a function that receives a generic object argument, the whatisThis method behaves according to the instance class thus implementing polymorphism.

[29]:
```
myvehicle = Vehicle()

def whatObject(obj):
    obj.whatisThis()

whatObject(myvehicle)
whatObject(mycar)
```

```
init vehicle
vehicle
car
```

### 3.0.4 A complete example with polymorphism and inheritance

[30]:
```
from math import pi

# base class definition
class Shape:
    # constructor for the base class: it simply assigns the name of the shape
    # that is passed by the derived classes constructor with the super object
    def __init__(self, name):
        self.name = name
```

```
    # this is an abstract method that MUST be filled in the children classes
    def area(self):
        pass

    # defines a printable version of class information
    def __str__(self):
        return self.name

# the Square class derives Shape and implements the required methods
class Square(Shape):
    # the super class gets initialized with the shape name and the
    # child one assigns the length of the square sides.
    def __init__(self, length):
        super().__init__("Square")
        self.length = length

    # area must be filled in the derived class as no implementation is
    # given in the base class being it an abstract method
    def area(self):
        return self.length**2

# another derived class for a Circle shape
class Circle(Shape):
    # the super class assigns the name of the shape and the radius
    # of a Circle object.
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    # area is defined according to the circle shape
    def area(self):
        return pi*self.radius**2
```

[31]:
```
# create two instances of our classes
a = Square(4)
b = Circle(3)

# iterate over the objects regardless of their classes and
# obtain the correct results due to polymorphism
for obj in (a,b):
    print('{} has area {:.2f}'.format(obj,obj.area()))
```

```
Square has area 16.00
Circle has area 28.27
```