module_1

May 9, 2023

0.1 The Python Imaging Library (PIL)

The Python Imaging Library, which is known as PIL or PILLOW, is the main library we use in python for dealing with image files. This library is not included with python - it's what's known as a third party library, which means you have to download and install it yourself. In the Coursera system, this has all been done for you. Lets do a little exploring of pillow in the jupyter notebooks.

```
In [ ]: # You'll recall that we import a library using the `import` keyword.
        import PIL
```

```
In [ ]: # Documentation is a big help in learning a library. There exist standards that make t
        # For example, most libraries let you check their version using the version attribute.
        PIL.__version__
```

```
In [ ]: # Let's figure out how to open an image with `Pillow`. Python provides some built-in f
        # understand the functions and objects which are available in libraries. For instance,
        # when called on any object, returns the objects built-in documentation. Lets try it w
        # module, PIL.
        help(PIL)
```

```
In [ ]: # This shows us that there are a host of classes available to us in the module, as wel
        # and even the file, called __init__.py, which has the source code for the module itse
        # the source code for this in the Jupyter console if we wanted to. These documentation
        # to poke around an unexplored library.
        #
        # Python also has a function called dir() which will list the contents of an object. T
        # with modules where you might want to see what classes you might interact with. Lets
        # the PIL module
        dir(PIL)
```

```
In [ ]: # At the top of the list, there is something called Image. This sounds like it could b
        # import it directly, and run the help command on it.
        from PIL import Image
        help(Image)
```

Running help() on Image tells us that this object is "the Image class wrapper". We see from the top level documentation about the image object that there is "hardly ever any reason to call the Image constructor directly", and they suggest that the open function might be the way to go.

```
In [ ]: # Lets call help on the open function to see what it's all about. Remember that since 
        # function reference, and not run the function itself, we don't put paretheses behind
        help(Image.open)
```

```
In [ ]: # It looks like Image.open() is a function that loads an image from a file and returns
        # of the Image class. Lets give it a try. In the read_only directory there is an image
        # which is from our Master's of Information program recruitment flyer. Lets try and lo

        file="readonly/msi_recruitment.gif"
        image=Image.open(file)
        print(image)
```

```
In [ ]: # Ok, we see that this returns us a kind of PIL.GifImagePlugin.GifImageFile. At first
        # seem a bit confusing, since because we were told by the docs that we should be exepc
        # PIL.Image.Image object back. But this is just object inheritance working! In fact, t
        # returned is both an Image and a GifImageFile. We can use the python inspect module t
        # as the getmro function will return a list of all of the classes that are being inher
        # given object. Lets try it.

        import inspect
        print("The type of the image is " + str(type(image)))
        inspect.getmro(type(image))
```

```
In [ ]: # Now that we are comfortable with the object. How do we view the image? It turns out
        # image object has a show function. You can find this by looking at all of the propert
        # the object if you wanted to, using the dir() function.
        image.show()
```

```
In [ ]: # Hrm, that didn't seem to have the intended effect. The problem is that the image is 
        # remotely, on Coursera's server, but show tries to show it locally to you. So, if the
        # server software was running on someone's workstation in Mountain View California, wh
        # has its offices, then you just popped up a picture of our recruitment materials. Tha
        # Instead, we want to render the image in the Jupyter notebook. It turns out Jupyter h
        # which can help with this.
        from IPython.display import display
        display(image)
```

For those who would like to understand this in more detail, the Jupyter environment is running a special wrapper around the Python interpretor, called IPython. IPython allows the kernel back end to communicate with a browser front end, among other things. The IPython package has a display function which can take objects and use custom formatters in order to render them. A number of formatters are provided by default, including one which knows how to handle image types.

That's a quick overview of how to read and display images using pillow, in the next lecture we'll jump in a bit more detail to understand how to use pillow to manipulate images.

0.2 Common Functions in the Python Imaging Library

Lets take a look at some of the common tasks we can do in python using the pillow library.

```
In [ ]: # First, lets import the PIL library and the Image object
        import PIL
        from PIL import Image
        # And lets import the display functionality
        from IPython.display import display
        # And finally, lets load the image we were working with last time
        file="readonly/msi_recruitment.gif"
        image=Image.open(file)

In [ ]: # Great, now lets check out a few more methods of the image library. First, we'll look
        # And if you remember, we can do this using the built in python help() function
        help(image.copy)

In [ ]:

In [ ]: # We can see that copy takes no arguments, and that the return object is an Image obje
        # look at save
        help(image.save)

In [ ]: # The save method has a couple of parameters which are interesting. The first, called
        # we want to save the object too. The second, format, is interesting, it allows us to
        # the image, but the docs tell us that this should be done automatically by looking at
        # as well. Lets give it a try -- this file was originally a GifImageFile, but I bet if
        # .png format and read it in again we'll get a different kind of file
        image.save("msi_recruitment.png")
        image=Image.open("msi_recruitment.png")
        import inspect
        inspect.getmro(type(image))

In [ ]: # Indeed, this created a new file, which we could view by going to the Jupyter noteboo
        # on the logo at the top of the browser, and we can see this new object is actually a
        # For the purposes of this class the difference in image formats isn't so important, b
        # explore how a library works using the functions of help(), dir() and getmro().
        #
        # The PILLOW library also has some nice image filters to add some effects. It does thi
        # function. The filter() function takes a Filter object, and those are all stored in t
        # Lets take a look.
        from PIL import ImageFilter
        help(ImageFilter)

In [ ]: # There are a bunch of different filters here, but lets just try and apply the BLUR fi
        # we have to convert the image to RGB mode. This is a bit magical -- images like gifs
        # colors can be displayed at once based on the size of the pallet. This is similar to
        # only has so much room. This is actually a very old image file format. If we convert
        # more sophisticated we can apply these interesting image transforms. Sometimes learni
        # digging a bit deeper into the domain the library is about. We can convert the image
        # function.
        image=image.convert('RGB') # this stands for red, green blue mode
        blurred_image=image.filter(PIL.ImageFilter.BLUR)
        display(blurred_image)
```

```
In [ ]: # Ok, let me show you one more function in the lecture, which is crop(). This removes
        # except for the bounding box you describe. When you think of images, think of individ
        # which make up that image being lined up in a grid. You can actually see the number o
        # is and the width of the image
        print("{}x{}".format(image.width, image.height))
```

```
In [ ]: # This means that the image is 800 pixels wide (the X axis), and 450 pixels high (the
        # look at the crop documentation we see that the first parameter to the function is a
        # left, upper, right, and lower values of the X/Y coordinates
        help(image.crop)
```

```
In [ ]: # With PIL images, we define the bounding box using the upper left corner and the lowe
        # we count the number of pixels out from the upper left corner, which is 0,0. This mig
        # used to coordinate systems where you start in the lower left -- just remember that w
        # same way we count out positions in the image.
        #
        # So, if we wanted to get the Michigan logo out of this image, we might start with the
        # and the top at 0 pixels, then we might walk to the right another 190 pixels, and set
        # 150 pixels
        display(image.crop((50,0,190,150)))
```

```
In [ ]: # Of course crop(), like other functions, only returns a copy of the image, and doesn'
        # A strategy I like to do is try and draw the bounding box directly on the image, when
        # up. We can draw on images using the ImageDraw object. I'm not going to go into this
        # quick example of how. I might draw the bounding box in this case.
        from PIL import ImageDraw
        drawing_object=ImageDraw.Draw(image)
        drawing_object.rectangle((50,0,190,150), fill = None, outline ='red')
        display(image)
```

Ok, that's been an overview of how to use PIL for single images. But, a lot of work might involve multiple images, and putting images together. In the next lecture we'll tackle that, and set you up for the assignment.

0.3 Additional PILLOW functions

Lets take a look at some other functions we might want to use in PILLOW to modify images.

```
In [ ]: # First, lets import all of the library functions we need
        import PIL
        from PIL import Image
        from IPython.display import display

        # And lets load the image we were working, and we can just convert it to RGB inline
        file="readonly/msi_recruitment.gif"
        image=Image.open(file).convert('RGB')

        display(image)
```

```
In [ ]: # First, lets import all of the library functions we need
        import PIL
        from PIL import Image
        from IPython.display import display

        # And lets load the image we were working, and we can just convert it to RGB inline
        file="readonly/msi_recruitment.gif"
        image=Image.open(file).convert('RGB')

        display(image)
```

```
In [ ]: # A task that is fairly common in image and picture manipulation is to create contact
        # A contact sheet is one image that actually contains several other different images.
        # a contact sheet for the Master of Science in Information advertisment image. In part
        # the brightness of the image in ten different ways, then scale the image down smaller
        # by side so we can get the sense of which brightness we might want to use.
        #
        # First up, lets import the ImageEnhance module, which has a nice object called Bright
        from PIL import ImageEnhance
        # Checking the online documentation for this function, it takes a value between 0.0 (a
        # image) and 1.0 (the original image) to adjust the brightness. All of the classes in
        # do this the same way, you create an object, in this case Brightness, then you call t
        # on that object with an appropriate parameter.
        #
        # Lets write a little loop to generate ten images of different brightness. First we ne
        # object with our image
        enhancer=ImageEnhance.Brightness(image)
        images=[]
        for i in range(0, 10):
            # We'll divide i by ten to get the decimal value we want, and append it to the ima
            # we actually call the brightness routine by calling the enhance() function. Remem
            # details of this using the help() function, or by consulting web docs
            images.append(enhancer.enhance(i/10))
        # We can see the result here is a list of ten PIL.Image.Image objects. Jupyter nicely
        # of python objects nested in lists
        print(images)
```

```
In [ ]: # Lets take these images now and composite them, one above another, in a contact sheet
        # There are several different approaches we can use, but I'll simply create a new imag
        # the first image, but ten times as high. Lets check out the PIL.Image.new functionali
        help(PIL.Image.new)
```

```
In [ ]: # The new function requires that we pass it a mode. We're going to use the mode 'RGB'
        # Red, Green, and Blue, and is the mode of our current first image. There are lots of
        # formats, and this one is most common.
        # For the size we have a tuple, which is the width of the image and the height. We'll
        # current first image, but for the height we'll multiple this by ten. This will make a
        # our contact sheet. Finally, the color is optional, and we'll just leave it at black.
```

```
         first_image=images[0]
         from PIL import Image
         contact_sheet=PIL.Image.new(first_image.mode, (first_image.width,10*first_image.height

         # So now we have a black image that's ten times the size of the other images in the co
         # variable. Now lets just loop through the image list and paste() the results in. The
         # will be called on the contact_sheet object, and takes in a new image to paste, as we
         # offset for that image. In our case, the x position is always 0, but the y location w
         # 450 pixels each time we iterate through the loop.
         #
         # Lets first create a counter variable for the y location. It will start at zero
         current_location = 0
         for img in images:
             # Lets paste the current image into the contact sheet
             contact_sheet.paste(img, (0, current_location) )
             # And update the current_location counter
             current_location=current_location+450

         # This contact sheet has gotten big: 4,500 pixels tall! Lets just resize this sheet fo
         # this using the resize() function. This function just takes a tuple of width and heig
         # everything down to the size of just two individual images
         contact_sheet = contact_sheet.resize((160,900) )
         # Now lets just display that composite image
         display(contact_sheet)

In [ ]: # Ok, that's a nice proof of concept. But it's a little tough to see. Lets instead cha
         # by three grid of values. First thing we should do is make our canvas, and we'll make
         # width of our image and 3 times the height of our image - a nine image square
         contact_sheet=PIL.Image.new(first_image.mode, (first_image.width*3,first_image.height*
         # Now we want to iterate over our images and place them into this grid. Remember that
         # location of where we refer to as an image in the upper right hand corner, so this wi
         # one variable for the X dimension, and one for the Y dimension.
         x=0
         y=0

         # Now, lets iterate over our images. Except, we don't want to both with the first one,
         # just solid black. Instead we want to just deal with the images after the first one,
         # give us nine in total
         for img in images[1:]:
             # Lets paste the current image into the contact sheet
             contact_sheet.paste(img, (x, y) )
             # Now we update our X position. If it is going to be the width of the image, then
             # and update Y as well to point to the next "line" of the contact sheet.
             if x+first_image.width == contact_sheet.width:
                 x=0
                 y=y+first_image.height
             else:
                 x=x+first_image.width
```

```
# Now lets resize the contact sheet. We'll just make it half the size by dividing it b
# the resize function needs to take round numbers, we need to convert our divisions fr
# numbers into integers using the int() function.
contact_sheet = contact_sheet.resize((int(contact_sheet.width/2),int(contact_sheet.hei
# Now lets display that composite image
display(contact_sheet)
```

Well, that's been a tour of our first external API, the Python Imaging Library, or pillow module. In this series of lectures you've learned how to read and write images, manipulat them with pillow, and explore the functionality of third party APIs using features of Python like dir(), help(), and getmro(). You've also been introduced to the console, and how python stores these libraries on the computer. While for this course all of the libraries are included for you in the Coursera system, and you won't need to install your own, it's good to get a the idea of how this work in case you wanted to set this up on your own.

Finally, while you can explore PILLOW from within python, most good modules also put their documentation up online, and you can read more about PILLOW here: https://pillow.readthedocs.io/en/latest/