module_1

July 12, 2020

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

```
In [2]: # 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__
```

```
Out[2]: '5.4.1'
```

```
In [3]: # 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)
```

```
Help on package PIL:

NAME
    PIL - Pillow (Fork of the Python Imaging Library)

DESCRIPTION
    Pillow is the friendly PIL fork by Alex Clark and Contributors.
        https://github.com/python-pillow/Pillow/

    Pillow is forked from PIL 1.1.7.

    PIL is the Python Imaging Library by Fredrik Lundh and Contributors.
    Copyright (c) 1999 by Secret Labs AB.

    Use PIL.__version__ for this Pillow version.
    PIL.VERSION is the old PIL version and will be removed in the future.
```

```
    ;-)

PACKAGE CONTENTS
    BdfFontFile
    BlpImagePlugin
    BmpImagePlugin
    BufrStubImagePlugin
    ContainerIO
    CurImagePlugin
    DcxImagePlugin
    DdsImagePlugin
    EpsImagePlugin
    ExifTags
    FitsStubImagePlugin
    FliImagePlugin
    FontFile
    FpxImagePlugin
    FtexImagePlugin
    GbrImagePlugin
    GdImageFile
    GifImagePlugin
    GimpGradientFile
    GimpPaletteFile
    GribStubImagePlugin
    Hdf5StubImagePlugin
    IcnsImagePlugin
    IcoImagePlugin
    ImImagePlugin
    Image
    ImageChops
    ImageCms
    ImageColor
    ImageDraw
    ImageDraw2
    ImageEnhance
    ImageFile
    ImageFilter
    ImageFont
    ImageGrab
    ImageMath
    ImageMode
    ImageMorph
    ImageOps
    ImagePalette
    ImagePath
    ImageQt
    ImageSequence
    ImageShow
```

```
ImageStat
ImageTk
ImageTransform
ImageWin
ImtImagePlugin
IptcImagePlugin
Jpeg2KImagePlugin
JpegImagePlugin
JpegPresets
McIdasImagePlugin
MicImagePlugin
MpegImagePlugin
MpoImagePlugin
MspImagePlugin
OleFileIO
PSDraw
PaletteFile
PalmImagePlugin
PcdImagePlugin
PcfFontFile
PcxImagePlugin
PdfImagePlugin
PdfParser
PixarImagePlugin
PngImagePlugin
PpmImagePlugin
PsdImagePlugin
PyAccess
SgiImagePlugin
SpiderImagePlugin
SunImagePlugin
TarIO
TgaImagePlugin
TiffImagePlugin
TiffTags
WalImageFile
WebPImagePlugin
WmfImagePlugin
XVThumbImagePlugin
XbmImagePlugin
XpmImagePlugin
_binary
_imaging
_imagingft
_imagingmath
_imagingmorph
_imagingtk
_tkinter_finder
```

```
    _util
    _version
    features

DATA
    PILLOW_VERSION = '5.4.1'
    VERSION = '1.1.7'

VERSION
    5.4.1

FILE
    /opt/conda/lib/python3.7/site-packages/PIL/__init__.py
```

In [4]:
```
# 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)
```

Out[4]:
```
['PILLOW_VERSION',
 'VERSION',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '__version__',
 '_plugins']
```

In [5]:
```
# 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)
```

```
Help on module PIL.Image in PIL:

NAME
```

```
    PIL.Image

DESCRIPTION
    # The Python Imaging Library.
    # $Id$
    #
    # the Image class wrapper
    #
    # partial release history:
    # 1995-09-09 fl   Created
    # 1996-03-11 fl   PIL release 0.0 (proof of concept)
    # 1996-04-30 fl   PIL release 0.1b1
    # 1999-07-28 fl   PIL release 1.0 final
    # 2000-06-07 fl   PIL release 1.1
    # 2000-10-20 fl   PIL release 1.1.1
    # 2001-05-07 fl   PIL release 1.1.2
    # 2002-03-15 fl   PIL release 1.1.3
    # 2003-05-10 fl   PIL release 1.1.4
    # 2005-03-28 fl   PIL release 1.1.5
    # 2006-12-02 fl   PIL release 1.1.6
    # 2009-11-15 fl   PIL release 1.1.7
    #
    # Copyright (c) 1997-2009 by Secret Labs AB.  All rights reserved.
    # Copyright (c) 1995-2009 by Fredrik Lundh.
    #
    # See the README file for information on usage and redistribution.
    #

CLASSES
    builtins.Exception(builtins.BaseException)
        DecompressionBombError
    builtins.RuntimeWarning(builtins.Warning)
        DecompressionBombWarning
    builtins.object
        Image
        ImagePointHandler
        ImageTransformHandler

    class DecompressionBombError(builtins.Exception)
     |  Common base class for all non-exit exceptions.
     |
     |  Method resolution order:
     |      DecompressionBombError
     |      builtins.Exception
     |      builtins.BaseException
     |      builtins.object
     |
     |  Data descriptors defined here:
```

```
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Methods inherited from builtins.Exception:
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Static methods inherited from builtins.Exception:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  ----------------------------------------------------------------------
|  Methods inherited from builtins.BaseException:
|
|  __delattr__(self, name, /)
|      Implement delattr(self, name).
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __reduce__(...)
|      Helper for pickle.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __setattr__(self, name, value, /)
|      Implement setattr(self, name, value).
|
|  __setstate__(...)
|
|  __str__(self, /)
|      Return str(self).
|
|  with_traceback(...)
|      Exception.with_traceback(tb) --
|      set self.__traceback__ to tb and return self.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from builtins.BaseException:
|
|  __cause__
|      exception cause
```

```
 |
 |  __context__
 |      exception context
 |
 |  __dict__
 |
 |  __suppress_context__
 |
 |  __traceback__
 |
 |  args

class DecompressionBombWarning(builtins.RuntimeWarning)
 |  Base class for warnings about dubious runtime behavior.
 |
 |  Method resolution order:
 |      DecompressionBombWarning
 |      builtins.RuntimeWarning
 |      builtins.Warning
 |      builtins.Exception
 |      builtins.BaseException
 |      builtins.object
 |
 |  Data descriptors defined here:
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.RuntimeWarning:
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Static methods inherited from builtins.RuntimeWarning:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from builtins.BaseException:
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
```

```
 |
 |  __reduce__(...)
 |      Helper for pickle.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  __setstate__(...)
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  with_traceback(...)
 |      Exception.with_traceback(tb) --
 |      set self.__traceback__ to tb and return self.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from builtins.BaseException:
 |
 |  __cause__
 |      exception cause
 |
 |  __context__
 |      exception context
 |
 |  __dict__
 |
 |  __suppress_context__
 |
 |  __traceback__
 |
 |  args

class Image(builtins.object)
 |  This class represents an image object.  To create
 |  :py:class:`~PIL.Image.Image` objects, use the appropriate factory
 |  functions.  There's hardly ever any reason to call the Image constructor
 |  directly.
 |
 |  * :py:func:`~PIL.Image.open`
 |  * :py:func:`~PIL.Image.new`
 |  * :py:func:`~PIL.Image.frombytes`
 |
 |  Methods defined here:
 |
```

```
|  __copy__ = copy(self)
|
|  __del__(self)
|
|  __enter__(self)
|      # Context manager support
|
|  __eq__(self, other)
|      Return self==value.
|
|  __exit__(self, *args)
|
|  __getstate__(self)
|
|  __init__(self)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __ne__(self, other)
|      Return self!=value.
|
|  __repr__(self)
|      Return repr(self).
|
|  __setstate__(self, state)
|
|  alpha_composite(self, im, dest=(0, 0), source=(0, 0))
|      'In-place' analog of Image.alpha_composite. Composites an image
|      onto this image.
|
|      :param im: image to composite over this one
|      :param dest: Optional 2 tuple (left, top) specifying the upper
|        left corner in this (destination) image.
|      :param source: Optional 2 (left, top) tuple for the upper left
|        corner in the overlay source image, or 4 tuple (left, top, right,
|        bottom) for the bounds of the source rectangle
|
|      Performance Note: Not currently implemented in-place in the core layer.
|
|  close(self)
|      Closes the file pointer, if possible.
|
|      This operation will destroy the image core and release its memory.
|      The image data will be unusable afterward.
|
|      This function is only required to close images that have not
|      had their file read and closed by the
|      :py:meth:`~PIL.Image.Image.load` method. See
|      :ref:`file-handling` for more information.
```

```
|
|  convert(self, mode=None, matrix=None, dither=None, palette=0, colors=256)
|      Returns a converted copy of this image. For the "P" mode, this
|      method translates pixels through the palette.  If mode is
|      omitted, a mode is chosen so that all information in the image
|      and the palette can be represented without a palette.
|
|      The current version supports all possible conversions between
|      "L", "RGB" and "CMYK." The **matrix** argument only supports "L"
|      and "RGB".
|
|      When translating a color image to greyscale (mode "L"),
|      the library uses the ITU-R 601-2 luma transform::
|
|          L = R * 299/1000 + G * 587/1000 + B * 114/1000
|
|      The default method of converting a greyscale ("L") or "RGB"
|      image into a bilevel (mode "1") image uses Floyd-Steinberg
|      dither to approximate the original image luminosity levels. If
|      dither is NONE, all values larger than 128 are set to 255 (white),
|      all other values to 0 (black). To use other thresholds, use the
|      :py:meth:`~PIL.Image.Image.point` method.
|
|      When converting from "RGBA" to "P" without a **matrix** argument,
|      this passes the operation to :py:meth:`~PIL.Image.Image.quantize`,
|      and **dither** and **palette** are ignored.
|
|      :param mode: The requested mode. See: :ref:`concept-modes`.
|      :param matrix: An optional conversion matrix.  If given, this
|         should be 4- or 12-tuple containing floating point values.
|      :param dither: Dithering method, used when converting from
|         mode "RGB" to "P" or from "RGB" or "L" to "1".
|         Available methods are NONE or FLOYDSTEINBERG (default).
|         Note that this is not used when **matrix** is supplied.
|      :param palette: Palette to use when converting from mode "RGB"
|         to "P".  Available palettes are WEB or ADAPTIVE.
|      :param colors: Number of colors to use for the ADAPTIVE palette.
|         Defaults to 256.
|      :rtype: :py:class:`~PIL.Image.Image`
|      :returns: An :py:class:`~PIL.Image.Image` object.
|
|  copy(self)
|      Copies this image. Use this method if you wish to paste things
|      into an image, but still retain the original.
|
|      :rtype: :py:class:`~PIL.Image.Image`
|      :returns: An :py:class:`~PIL.Image.Image` object.
|
```

```
|  crop(self, box=None)
|      Returns a rectangular region from this image. The box is a
|      4-tuple defining the left, upper, right, and lower pixel
|      coordinate. See :ref:`coordinate-system`.
|
|      Note: Prior to Pillow 3.4.0, this was a lazy operation.
|
|      :param box: The crop rectangle, as a (left, upper, right, lower)-tuple.
|      :rtype: :py:class:`~PIL.Image.Image`
|      :returns: An :py:class:`~PIL.Image.Image` object.
|
|  draft(self, mode, size)
|      Configures the image file loader so it returns a version of the
|      image that as closely as possible matches the given mode and
|      size.  For example, you can use this method to convert a color
|      JPEG to greyscale while loading it, or to extract a 128x192
|      version from a PCD file.
|
|      Note that this method modifies the :py:class:`~PIL.Image.Image` object
|      in place.  If the image has already been loaded, this method has no
|      effect.
|
|      Note: This method is not implemented for most images. It is
|      currently implemented only for JPEG and PCD images.
|
|      :param mode: The requested mode.
|      :param size: The requested size.
|
|  effect_spread(self, distance)
|      Randomly spread pixels in an image.
|
|      :param distance: Distance to spread pixels.
|
|  filter(self, filter)
|      Filters this image using the given filter.  For a list of
|      available filters, see the :py:mod:`~PIL.ImageFilter` module.
|
|      :param filter: Filter kernel.
|      :returns: An :py:class:`~PIL.Image.Image` object.
|
|  frombytes(self, data, decoder_name='raw', *args)
|      Loads this image with pixel data from a bytes object.
|
|      This method is similar to the :py:func:`~PIL.Image.frombytes` function,
|      but loads data into this image instead of creating a new image object.
|
|  fromstring(self, *args, **kw)
|
```

```
|  getbands(self)
|      Returns a tuple containing the name of each band in this image.
|      For example, **getbands** on an RGB image returns ("R", "G", "B").
|
|      :returns: A tuple containing band names.
|      :rtype: tuple
|
|  getbbox(self)
|      Calculates the bounding box of the non-zero regions in the
|      image.
|
|      :returns: The bounding box is returned as a 4-tuple defining the
|         left, upper, right, and lower pixel coordinate. See
|         :ref:`coordinate-system`. If the image is completely empty, this
|         method returns None.
|
|  getchannel(self, channel)
|      Returns an image containing a single channel of the source image.
|
|      :param channel: What channel to return. Could be index
|        (0 for "R" channel of "RGB") or channel name
|        ("A" for alpha channel of "RGBA").
|      :returns: An image in "L" mode.
|
|      .. versionadded:: 4.3.0
|
|  getcolors(self, maxcolors=256)
|      Returns a list of colors used in this image.
|
|      :param maxcolors: Maximum number of colors.  If this number is
|         exceeded, this method returns None.  The default limit is
|         256 colors.
|      :returns: An unsorted list of (count, pixel) values.
|
|  getdata(self, band=None)
|      Returns the contents of this image as a sequence object
|      containing pixel values.  The sequence object is flattened, so
|      that values for line one follow directly after the values of
|      line zero, and so on.
|
|      Note that the sequence object returned by this method is an
|      internal PIL data type, which only supports certain sequence
|      operations.  To convert it to an ordinary sequence (e.g. for
|      printing), use **list(im.getdata())**.
|
|      :param band: What band to return.  The default is to return
|         all bands.  To return a single band, pass in the index
|         value (e.g. 0 to get the "R" band from an "RGB" image).
```

```
|      :returns: A sequence-like object.
|
|  getextrema(self)
|      Gets the the minimum and maximum pixel values for each band in
|      the image.
|
|      :returns: For a single-band image, a 2-tuple containing the
|         minimum and maximum pixel value.  For a multi-band image,
|         a tuple containing one 2-tuple for each band.
|
|  getim(self)
|      Returns a capsule that points to the internal image memory.
|
|      :returns: A capsule object.
|
|  getpalette(self)
|      Returns the image palette as a list.
|
|      :returns: A list of color values [r, g, b, ...], or None if the
|         image has no palette.
|
|  getpixel(self, xy)
|      Returns the pixel value at a given position.
|
|      :param xy: The coordinate, given as (x, y). See
|         :ref:`coordinate-system`.
|      :returns: The pixel value.  If the image is a multi-layer image,
|         this method returns a tuple.
|
|  getprojection(self)
|      Get projection to x and y axes
|
|      :returns: Two sequences, indicating where there are non-zero
|          pixels along the X-axis and the Y-axis, respectively.
|
|  histogram(self, mask=None, extrema=None)
|      Returns a histogram for the image. The histogram is returned as
|      a list of pixel counts, one for each pixel value in the source
|      image. If the image has more than one band, the histograms for
|      all bands are concatenated (for example, the histogram for an
|      "RGB" image contains 768 values).
|
|      A bilevel image (mode "1") is treated as a greyscale ("L") image
|      by this method.
|
|      If a mask is provided, the method returns a histogram for those
|      parts of the image where the mask image is non-zero. The mask
|      image must have the same size as the image, and be either a
```

```
|      bi-level image (mode "1") or a greyscale image ("L").
|
|      :param mask: An optional mask.
|      :returns: A list containing pixel counts.
|
|  load(self)
|      Allocates storage for the image and loads the pixel data.  In
|      normal cases, you don't need to call this method, since the
|      Image class automatically loads an opened image when it is
|      accessed for the first time.
|
|      If the file associated with the image was opened by Pillow, then this
|      method will close it. The exception to this is if the image has
|      multiple frames, in which case the file will be left open for seek
|      operations. See :ref:`file-handling` for more information.
|
|      :returns: An image access object.
|      :rtype: :ref:`PixelAccess` or :py:class:`PIL.PyAccess`
|
|  offset(self, xoffset, yoffset=None)
|
|  paste(self, im, box=None, mask=None)
|      Pastes another image into this image. The box argument is either
|      a 2-tuple giving the upper left corner, a 4-tuple defining the
|      left, upper, right, and lower pixel coordinate, or None (same as
|      (0, 0)). See :ref:`coordinate-system`. If a 4-tuple is given, the size
|      of the pasted image must match the size of the region.
|
|      If the modes don't match, the pasted image is converted to the mode of
|      this image (see the :py:meth:`~PIL.Image.Image.convert` method for
|      details).
|
|      Instead of an image, the source can be a integer or tuple
|      containing pixel values.  The method then fills the region
|      with the given color.  When creating RGB images, you can
|      also use color strings as supported by the ImageColor module.
|
|      If a mask is given, this method updates only the regions
|      indicated by the mask.  You can use either "1", "L" or "RGBA"
|      images (in the latter case, the alpha band is used as mask).
|      Where the mask is 255, the given image is copied as is.  Where
|      the mask is 0, the current value is preserved.  Intermediate
|      values will mix the two images together, including their alpha
|      channels if they have them.
|
|      See :py:meth:`~PIL.Image.Image.alpha_composite` if you want to
|      combine images with respect to their alpha channels.
|
```

```
|      :param im: Source image or pixel value (integer or tuple).
|      :param box: An optional 4-tuple giving the region to paste into.
|         If a 2-tuple is used instead, it's treated as the upper left
|         corner.  If omitted or None, the source is pasted into the
|         upper left corner.
|
|         If an image is given as the second argument and there is no
|         third, the box defaults to (0, 0), and the second argument
|         is interpreted as a mask image.
|      :param mask: An optional mask image.
|
|  point(self, lut, mode=None)
|      Maps this image through a lookup table or function.
|
|      :param lut: A lookup table, containing 256 (or 65536 if
|         self.mode=="I" and mode == "L") values per band in the
|         image.  A function can be used instead, it should take a
|         single argument. The function is called once for each
|         possible pixel value, and the resulting table is applied to
|         all bands of the image.
|      :param mode: Output mode (default is same as input).  In the
|         current version, this can only be used if the source image
|         has mode "L" or "P", and the output has mode "1" or the
|         source image mode is "I" and the output mode is "L".
|      :returns: An :py:class:`~PIL.Image.Image` object.
|
|  putalpha(self, alpha)
|      Adds or replaces the alpha layer in this image.  If the image
|      does not have an alpha layer, it's converted to "LA" or "RGBA".
|      The new layer must be either "L" or "1".
|
|      :param alpha: The new alpha layer.  This can either be an "L" or "1"
|         image having the same size as this image, or an integer or
|         other color value.
|
|  putdata(self, data, scale=1.0, offset=0.0)
|      Copies pixel data to this image.  This method copies data from a
|      sequence object into the image, starting at the upper left
|      corner (0, 0), and continuing until either the image or the
|      sequence ends.  The scale and offset values are used to adjust
|      the sequence values: **pixel = value*scale + offset**.
|
|      :param data: A sequence object.
|      :param scale: An optional scale value.  The default is 1.0.
|      :param offset: An optional offset value.  The default is 0.0.
|
|  putpalette(self, data, rawmode='RGB')
|      Attaches a palette to this image.  The image must be a "P" or
```

```
|      "L" image, and the palette sequence must contain 768 integer
|      values, where each group of three values represent the red,
|      green, and blue values for the corresponding pixel
|      index. Instead of an integer sequence, you can use an 8-bit
|      string.
|
|      :param data: A palette sequence (either a list or a string).
|      :param rawmode: The raw mode of the palette.
|
|  putpixel(self, xy, value)
|      Modifies the pixel at the given position. The color is given as
|      a single numerical value for single-band images, and a tuple for
|      multi-band images. In addition to this, RGB and RGBA tuples are
|      accepted for P images.
|
|      Note that this method is relatively slow.  For more extensive changes,
|      use :py:meth:`~PIL.Image.Image.paste` or the :py:mod:`~PIL.ImageDraw`
|      module instead.
|
|      See:
|
|      * :py:meth:`~PIL.Image.Image.paste`
|      * :py:meth:`~PIL.Image.Image.putdata`
|      * :py:mod:`~PIL.ImageDraw`
|
|      :param xy: The pixel coordinate, given as (x, y). See
|         :ref:`coordinate-system`.
|      :param value: The pixel value.
|
|  quantize(self, colors=256, method=None, kmeans=0, palette=None)
|      Convert the image to 'P' mode with the specified number
|      of colors.
|
|      :param colors: The desired number of colors, <= 256
|      :param method: 0 = median cut
|                     1 = maximum coverage
|                     2 = fast octree
|                     3 = libimagequant
|      :param kmeans: Integer
|      :param palette: Quantize to the palette of given
|                      :py:class:`PIL.Image.Image`.
|      :returns: A new image
|
|  remap_palette(self, dest_map, source_palette=None)
|      Rewrites the image to reorder the palette.
|
|      :param dest_map: A list of indexes into the original palette.
|         e.g. [1,0] would swap a two item palette, and list(range(255))
```

```
|         is the identity transform.
|      :param source_palette: Bytes or None.
|      :returns:  An :py:class:`~PIL.Image.Image` object.
|
|  resize(self, size, resample=0, box=None)
|      Returns a resized copy of this image.
|
|      :param size: The requested size in pixels, as a 2-tuple:
|         (width, height).
|      :param resample: An optional resampling filter.  This can be
|         one of :py:attr:`PIL.Image.NEAREST`, :py:attr:`PIL.Image.BOX`,
|         :py:attr:`PIL.Image.BILINEAR`, :py:attr:`PIL.Image.HAMMING`,
|         :py:attr:`PIL.Image.BICUBIC` or :py:attr:`PIL.Image.LANCZOS`.
|         If omitted, or if the image has mode "1" or "P", it is
|         set :py:attr:`PIL.Image.NEAREST`.
|         See: :ref:`concept-filters`.
|      :param box: An optional 4-tuple of floats giving the region
|         of the source image which should be scaled.
|         The values should be within (0, 0, width, height) rectangle.
|         If omitted or None, the entire source is used.
|      :returns: An :py:class:`~PIL.Image.Image` object.
|
|  rotate(self, angle, resample=0, expand=0, center=None, translate=None, fillcolor=None)
|      Returns a rotated copy of this image.  This method returns a
|      copy of this image, rotated the given number of degrees counter
|      clockwise around its centre.
|
|      :param angle: In degrees counter clockwise.
|      :param resample: An optional resampling filter.  This can be
|         one of :py:attr:`PIL.Image.NEAREST` (use nearest neighbour),
|         :py:attr:`PIL.Image.BILINEAR` (linear interpolation in a 2x2
|         environment), or :py:attr:`PIL.Image.BICUBIC`
|         (cubic spline interpolation in a 4x4 environment).
|         If omitted, or if the image has mode "1" or "P", it is
|         set :py:attr:`PIL.Image.NEAREST`. See :ref:`concept-filters`.
|      :param expand: Optional expansion flag.  If true, expands the output
|         image to make it large enough to hold the entire rotated image.
|         If false or omitted, make the output image the same size as the
|         input image.  Note that the expand flag assumes rotation around
|         the center and no translation.
|      :param center: Optional center of rotation (a 2-tuple).  Origin is
|         the upper left corner.  Default is the center of the image.
|      :param translate: An optional post-rotate translation (a 2-tuple).
|      :param fillcolor: An optional color for area outside the rotated image.
|      :returns: An :py:class:`~PIL.Image.Image` object.
|
|  save(self, fp, format=None, **params)
|      Saves this image under the given filename.  If no format is
```

```
|      specified, the format to use is determined from the filename
|      extension, if possible.
|
|      Keyword options can be used to provide additional instructions
|      to the writer. If a writer doesn't recognise an option, it is
|      silently ignored. The available options are described in the
|      :doc:`image format documentation
|      <../handbook/image-file-formats>` for each writer.
|
|      You can use a file object instead of a filename. In this case,
|      you must always specify the format. The file object must
|      implement the ``seek``, ``tell``, and ``write``
|      methods, and be opened in binary mode.
|
|      :param fp: A filename (string), pathlib.Path object or file object.
|      :param format: Optional format override.  If omitted, the
|         format to use is determined from the filename extension.
|         If a file object was used instead of a filename, this
|         parameter should always be used.
|      :param params: Extra parameters to the image writer.
|      :returns: None
|      :exception ValueError: If the output format could not be determined
|         from the file name.  Use the format option to solve this.
|      :exception IOError: If the file could not be written.  The file
|         may have been created, and may contain partial data.
|
|  seek(self, frame)
|      Seeks to the given frame in this sequence file. If you seek
|      beyond the end of the sequence, the method raises an
|      **EOFError** exception. When a sequence file is opened, the
|      library automatically seeks to frame 0.
|
|      Note that in the current version of the library, most sequence
|      formats only allows you to seek to the next frame.
|
|      See :py:meth:`~PIL.Image.Image.tell`.
|
|      :param frame: Frame number, starting at 0.
|      :exception EOFError: If the call attempts to seek beyond the end
|          of the sequence.
|
|  show(self, title=None, command=None)
|      Displays this image. This method is mainly intended for
|      debugging purposes.
|
|      On Unix platforms, this method saves the image to a temporary
|      PPM file, and calls either the **xv** utility or the **display**
|      utility, depending on which one can be found.
```

```
|
|      On macOS, this method saves the image to a temporary BMP file, and
|      opens it with the native Preview application.
|
|      On Windows, it saves the image to a temporary BMP file, and uses
|      the standard BMP display utility to show it (usually Paint).
|
|      :param title: Optional title to use for the image window,
|         where possible.
|      :param command: command used to show the image
|
|  split(self)
|      Split this image into individual bands. This method returns a
|      tuple of individual image bands from an image. For example,
|      splitting an "RGB" image creates three new images each
|      containing a copy of one of the original bands (red, green,
|      blue).
|
|      If you need only one band, :py:meth:`~PIL.Image.Image.getchannel`
|      method can be more convenient and faster.
|
|      :returns: A tuple containing bands.
|
|  tell(self)
|      Returns the current frame number. See :py:meth:`~PIL.Image.Image.seek`.
|
|      :returns: Frame number, starting with 0.
|
|  thumbnail(self, size, resample=3)
|      Make this image into a thumbnail.  This method modifies the
|      image to contain a thumbnail version of itself, no larger than
|      the given size.  This method calculates an appropriate thumbnail
|      size to preserve the aspect of the image, calls the
|      :py:meth:`~PIL.Image.Image.draft` method to configure the file reader
|      (where applicable), and finally resizes the image.
|
|      Note that this function modifies the :py:class:`~PIL.Image.Image`
|      object in place.  If you need to use the full resolution image as well,
|      apply this method to a :py:meth:`~PIL.Image.Image.copy` of the original
|      image.
|
|      :param size: Requested size.
|      :param resample: Optional resampling filter.  This can be one
|         of :py:attr:`PIL.Image.NEAREST`, :py:attr:`PIL.Image.BILINEAR`,
|         :py:attr:`PIL.Image.BICUBIC`, or :py:attr:`PIL.Image.LANCZOS`.
|         If omitted, it defaults to :py:attr:`PIL.Image.BICUBIC`.
|         (was :py:attr:`PIL.Image.NEAREST` prior to version 2.5.0)
|      :returns: None
```

```
|
|  tobitmap(self, name='image')
|      Returns the image converted to an X11 bitmap.
|
|      .. note:: This method only works for mode "1" images.
|
|      :param name: The name prefix to use for the bitmap variables.
|      :returns: A string containing an X11 bitmap.
|      :raises ValueError: If the mode is not "1"
|
|  tobytes(self, encoder_name='raw', *args)
|      Return image as a bytes object.
|
|      .. warning::
|
|          This method returns the raw image data from the internal
|          storage.  For compressed image data (e.g. PNG, JPEG) use
|          :meth:`~.save`, with a BytesIO parameter for in-memory
|          data.
|
|      :param encoder_name: What encoder to use.  The default is to
|                           use the standard "raw" encoder.
|      :param args: Extra arguments to the encoder.
|      :rtype: A bytes object.
|
|  toqimage(self)
|      Returns a QImage copy of this image
|
|  toqpixmap(self)
|      Returns a QPixmap copy of this image
|
|  tostring(self, *args, **kw)
|
|  transform(self, size, method, data=None, resample=0, fill=1, fillcolor=None)
|      Transforms this image.  This method creates a new image with the
|      given size, and the same mode as the original, and copies data
|      to the new image using the given transform.
|
|      :param size: The output size.
|      :param method: The transformation method.  This is one of
|        :py:attr:`PIL.Image.EXTENT` (cut out a rectangular subregion),
|        :py:attr:`PIL.Image.AFFINE` (affine transform),
|        :py:attr:`PIL.Image.PERSPECTIVE` (perspective transform),
|        :py:attr:`PIL.Image.QUAD` (map a quadrilateral to a rectangle), or
|        :py:attr:`PIL.Image.MESH` (map a number of source quadrilaterals
|        in one operation).
|
|        It may also be an :py:class:`~PIL.Image.ImageTransformHandler`
```

```
|       object::
|         class Example(Image.ImageTransformHandler):
|             def transform(size, method, data, resample, fill=1):
|                 # Return result
|
|       It may also be an object with a :py:meth:`~method.getdata` method
|       that returns a tuple supplying new **method** and **data** values::
|         class Example(object):
|             def getdata(self):
|                 method = Image.EXTENT
|                 data = (0, 0, 100, 100)
|                 return method, data
|     :param data: Extra data to the transformation method.
|     :param resample: Optional resampling filter.  It can be one of
|        :py:attr:`PIL.Image.NEAREST` (use nearest neighbour),
|        :py:attr:`PIL.Image.BILINEAR` (linear interpolation in a 2x2
|        environment), or :py:attr:`PIL.Image.BICUBIC` (cubic spline
|        interpolation in a 4x4 environment). If omitted, or if the image
|        has mode "1" or "P", it is set to :py:attr:`PIL.Image.NEAREST`.
|     :param fill: If **method** is an
|       :py:class:`~PIL.Image.ImageTransformHandler` object, this is one of
|       the arguments passed to it. Otherwise, it is unused.
|     :param fillcolor: Optional fill color for the area outside the
|        transform in the output image.
|     :returns: An :py:class:`~PIL.Image.Image` object.
|
|  transpose(self, method)
|      Transpose image (flip or rotate in 90 degree steps)
|
|      :param method: One of :py:attr:`PIL.Image.FLIP_LEFT_RIGHT`,
|        :py:attr:`PIL.Image.FLIP_TOP_BOTTOM`, :py:attr:`PIL.Image.ROTATE_90`,
|        :py:attr:`PIL.Image.ROTATE_180`, :py:attr:`PIL.Image.ROTATE_270`,
|        :py:attr:`PIL.Image.TRANSPOSE` or :py:attr:`PIL.Image.TRANSVERSE`.
|      :returns: Returns a flipped or rotated copy of this image.
|
|  verify(self)
|      Verifies the contents of a file. For data read from a file, this
|      method attempts to determine if the file is broken, without
|      actually decoding the image data.  If this method finds any
|      problems, it raises suitable exceptions.  If you need to load
|      the image after using this method, you must reopen the image
|      file.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __array_interface__
|
```

```
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  height
 |
 |  size
 |
 |  width
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
 |
 |  format = None
 |
 |  format_description = None

class ImagePointHandler(builtins.object)
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class ImageTransformHandler(builtins.object)
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

FUNCTIONS
    alpha_composite(im1, im2)
        Alpha composite im2 over im1.

        :param im1: The first image. Must have mode RGBA.
        :param im2: The second image.  Must have mode RGBA, and the same size as
           the first image.
        :returns: An :py:class:`~PIL.Image.Image` object.
```

```
blend(im1, im2, alpha)
    Creates a new image by interpolating between two input images, using
    a constant alpha.::

        out = image1 * (1.0 - alpha) + image2 * alpha

    :param im1: The first image.
    :param im2: The second image.  Must have the same mode and size as
       the first image.
    :param alpha: The interpolation alpha factor.  If alpha is 0.0, a
       copy of the first image is returned. If alpha is 1.0, a copy of
       the second image is returned. There are no restrictions on the
       alpha value. If necessary, the result is clipped to fit into
       the allowed output range.
    :returns: An :py:class:`~PIL.Image.Image` object.

coerce_e(value)

composite(image1, image2, mask)
    Create composite image by blending images using a transparency mask.

    :param image1: The first image.
    :param image2: The second image.  Must have the same mode and
       size as the first image.
    :param mask: A mask image.  This image can have mode
       "1", "L", or "RGBA", and must have the same size as the
       other two images.

effect_mandelbrot(size, extent, quality)
    Generate a Mandelbrot set covering the given extent.

    :param size: The requested size in pixels, as a 2-tuple:
       (width, height).
    :param extent: The extent to cover, as a 4-tuple:
       (x0, y0, x1, y2).
    :param quality: Quality.

effect_noise(size, sigma)
    Generate Gaussian noise centered around 128.

    :param size: The requested size in pixels, as a 2-tuple:
       (width, height).
    :param sigma: Standard deviation of noise.

eval(image, *args)
    Applies the function (which should take one argument) to each pixel
    in the given image. If the image has more than one band, the same
    function is applied to each band. Note that the function is
```

```
    evaluated once for each possible pixel value, so you cannot use
    random components or other generators.

    :param image: The input image.
    :param function: A function object, taking one integer argument.
    :returns: An :py:class:`~PIL.Image.Image` object.

fromarray(obj, mode=None)
    Creates an image memory from an object exporting the array interface
    (using the buffer protocol).

    If **obj** is not contiguous, then the tobytes method is called
    and :py:func:`~PIL.Image.frombuffer` is used.

    If you have an image in NumPy::

      from PIL import Image
      import numpy as np
      im = Image.open('hopper.jpg')
      a = np.asarray(im)

    Then this can be used to convert it to a Pillow image::

      im = Image.fromarray(a)

    :param obj: Object with array interface
    :param mode: Mode to use (will be determined from type if None)
      See: :ref:`concept-modes`.
    :returns: An image object.

    .. versionadded:: 1.1.6

frombuffer(mode, size, data, decoder_name='raw', *args)
    Creates an image memory referencing pixel data in a byte buffer.

    This function is similar to :py:func:`~PIL.Image.frombytes`, but uses data
    in the byte buffer, where possible.  This means that changes to the
    original buffer object are reflected in this image).  Not all modes can
    share memory; supported modes include "L", "RGBX", "RGBA", and "CMYK".

    Note that this function decodes pixel data only, not entire images.
    If you have an entire image file in a string, wrap it in a
    **BytesIO** object, and use :py:func:`~PIL.Image.open` to load it.

    In the current version, the default parameters used for the "raw" decoder
    differs from that used for :py:func:`~PIL.Image.frombytes`.  This is a
    bug, and will probably be fixed in a future release.  The current release
    issues a warning if you do this; to disable the warning, you should provide
```

```
    the full set of parameters.  See below for details.

    :param mode: The image mode. See: :ref:`concept-modes`.
    :param size: The image size.
    :param data: A bytes or other buffer object containing raw
        data for the given mode.
    :param decoder_name: What decoder to use.
    :param args: Additional parameters for the given decoder.  For the
        default encoder ("raw"), it's recommended that you provide the
        full set of parameters::

            frombuffer(mode, size, data, "raw", mode, 0, 1)

    :returns: An :py:class:`~PIL.Image.Image` object.

    .. versionadded:: 1.1.4

frombytes(mode, size, data, decoder_name='raw', *args)
    Creates a copy of an image memory from pixel data in a buffer.

    In its simplest form, this function takes three arguments
    (mode, size, and unpacked pixel data).

    You can also use any pixel decoder supported by PIL.  For more
    information on available decoders, see the section
    :ref:`Writing Your Own File Decoder <file-decoders>`.

    Note that this function decodes pixel data only, not entire images.
    If you have an entire image in a string, wrap it in a
    :py:class:`~io.BytesIO` object, and use :py:func:`~PIL.Image.open` to load
    it.

    :param mode: The image mode. See: :ref:`concept-modes`.
    :param size: The image size.
    :param data: A byte buffer containing raw data for the given mode.
    :param decoder_name: What decoder to use.
    :param args: Additional parameters for the given decoder.
    :returns: An :py:class:`~PIL.Image.Image` object.

fromqimage(im)
    Creates an image instance from a QImage image

fromqpixmap(im)
    Creates an image instance from a QPixmap image

fromstring(*args, **kw)

getmodebandnames(mode)
```

```
    Gets a list of individual band names.  Given a mode, this function returns
    a tuple containing the names of individual bands (use
    :py:method:`~PIL.Image.getmodetype` to get the mode used to store each
    individual band.

    :param mode: Input mode.
    :returns: A tuple containing band names.  The length of the tuple
        gives the number of bands in an image of the given mode.
    :exception KeyError: If the input mode was not a standard mode.

getmodebands(mode)
    Gets the number of individual bands for this mode.

    :param mode: Input mode.
    :returns: The number of bands in this mode.
    :exception KeyError: If the input mode was not a standard mode.

getmodebase(mode)
    Gets the "base" mode for given mode.  This function returns "L" for
    images that contain grayscale data, and "RGB" for images that
    contain color data.

    :param mode: Input mode.
    :returns: "L" or "RGB".
    :exception KeyError: If the input mode was not a standard mode.

getmodetype(mode)
    Gets the storage type mode.  Given a mode, this function returns a
    single-layer mode suitable for storing individual bands.

    :param mode: Input mode.
    :returns: "L", "I", or "F".
    :exception KeyError: If the input mode was not a standard mode.

init()
    Explicitly initializes the Python Imaging Library. This function
    loads all available file format drivers.

isImageType(t)
    Checks if an object is an image object.

    .. warning::

       This function is for internal use only.

    :param t: object to check if it's an image
    :returns: True if the object is an image
```

```
linear_gradient(mode)
    Generate 256x256 linear gradient from black to white, top to bottom.

    :param mode: Input mode.

merge(mode, bands)
    Merge a set of single band images into a new multiband image.

    :param mode: The mode to use for the output image. See:
        :ref:`concept-modes`.
    :param bands: A sequence containing one single-band image for
        each band in the output image.  All bands must have the
        same size.
    :returns: An :py:class:`~PIL.Image.Image` object.

new(mode, size, color=0)
    Creates a new image with the given mode and size.

    :param mode: The mode to use for the new image. See:
       :ref:`concept-modes`.
    :param size: A 2-tuple, containing (width, height) in pixels.
    :param color: What color to use for the image.  Default is black.
       If given, this should be a single integer or floating point value
       for single-band modes, and a tuple for multi-band modes (one value
       per band).  When creating RGB images, you can also use color
       strings as supported by the ImageColor module.  If the color is
       None, the image is not initialised.
    :returns: An :py:class:`~PIL.Image.Image` object.

open(fp, mode='r')
    Opens and identifies the given image file.

    This is a lazy operation; this function identifies the file, but
    the file remains open and the actual image data is not read from
    the file until you try to process the data (or call the
    :py:meth:`~PIL.Image.Image.load` method).  See
    :py:func:`~PIL.Image.new`. See :ref:`file-handling`.

    :param fp: A filename (string), pathlib.Path object or a file object.
       The file object must implement :py:meth:`~file.read`,
       :py:meth:`~file.seek`, and :py:meth:`~file.tell` methods,
       and be opened in binary mode.
    :param mode: The mode.  If given, this argument must be "r".
    :returns: An :py:class:`~PIL.Image.Image` object.
    :exception IOError: If the file cannot be found, or the image cannot be
       opened and identified.

preinit()
```

```
    Explicitly load standard file format drivers.

radial_gradient(mode)
    Generate 256x256 radial gradient from black to white, centre to edge.

    :param mode: Input mode.

register_decoder(name, decoder)
    Registers an image decoder.  This function should not be
    used in application code.

    :param name: The name of the decoder
    :param decoder: A callable(mode, args) that returns an
                    ImageFile.PyDecoder object

    .. versionadded:: 4.1.0

register_encoder(name, encoder)
    Registers an image encoder.  This function should not be
    used in application code.

    :param name: The name of the encoder
    :param encoder: A callable(mode, args) that returns an
                    ImageFile.PyEncoder object

    .. versionadded:: 4.1.0

register_extension(id, extension)
    Registers an image extension.  This function should not be
    used in application code.

    :param id: An image format identifier.
    :param extension: An extension used for this format.

register_extensions(id, extensions)
    Registers image extensions.  This function should not be
    used in application code.

    :param id: An image format identifier.
    :param extensions: A list of extensions used for this format.

register_mime(id, mimetype)
    Registers an image MIME type.  This function should not be used
    in application code.

    :param id: An image format identifier.
    :param mimetype: The image MIME type for this format.
```

```
    register_open(id, factory, accept=None)
        Register an image file plugin.  This function should not be used
        in application code.

        :param id: An image format identifier.
        :param factory: An image file factory method.
        :param accept: An optional function that can be used to quickly
           reject images having another format.

    register_save(id, driver)
        Registers an image save function.  This function should not be
        used in application code.

        :param id: An image format identifier.
        :param driver: A function to save images in this format.

    register_save_all(id, driver)
        Registers an image function to save all the frames
        of a multiframe format.  This function should not be
        used in application code.

        :param id: An image format identifier.
        :param driver: A function to save images in this format.

    registered_extensions()
        Returns a dictionary containing all file extensions belonging
        to registered plugins

DATA
    ADAPTIVE = 1
    AFFINE = 0
    ANTIALIAS = 1
    BICUBIC = 3
    BILINEAR = 2
    BOX = 4
    CONTAINER = 2
    CUBIC = 3
    DECODERS = {}
    DEFAULT_STRATEGY = 0
    ENCODERS = {}
    EXTENSION = {}
    EXTENT = 1
    FASTOCTREE = 2
    FILTERED = 1
    FIXED = 4
    FLIP_LEFT_RIGHT = 0
    FLIP_TOP_BOTTOM = 1
    FLOYDSTEINBERG = 3
```

```
    HAMMING = 5
    HAS_PATHLIB = True
    HUFFMAN_ONLY = 2
    ID = []
    LANCZOS = 1
    LIBIMAGEQUANT = 3
    LINEAR = 2
    MAXCOVERAGE = 1
    MAX_IMAGE_PIXELS = 89478485
    MEDIANCUT = 0
    MESH = 4
    MIME = {}
    MODES = ['1', 'CMYK', 'F', 'HSV', 'I', 'L', 'LAB', 'P', 'RGB', 'RGBA',...
    NEAREST = 0
    NONE = 0
    NORMAL = 0
    OPEN = {}
    ORDERED = 1
    PERSPECTIVE = 2
    PILLOW_VERSION = '5.4.1'
    QUAD = 3
    RASTERIZE = 2
    RLE = 3
    ROTATE_180 = 3
    ROTATE_270 = 4
    ROTATE_90 = 2
    SAVE = {}
    SAVE_ALL = {}
    SEQUENCE = 1
    TRANSPOSE = 5
    TRANSVERSE = 6
    USE_CFFI_ACCESS = False
    VERSION = '1.1.7'
    WEB = 0
    logger = <Logger PIL.Image (WARNING)>
    py3 = True

VERSION
    5.4.1

FILE
    /opt/conda/lib/python3.7/site-packages/PIL/Image.py
```

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 [6]: # 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)
```

```
Help on function open in module PIL.Image:

open(fp, mode='r')
    Opens and identifies the given image file.

    This is a lazy operation; this function identifies the file, but
    the file remains open and the actual image data is not read from
    the file until you try to process the data (or call the
    :py:meth:`~PIL.Image.Image.load` method).  See
    :py:func:`~PIL.Image.new`. See :ref:`file-handling`.

    :param fp: A filename (string), pathlib.Path object or a file object.
       The file object must implement :py:meth:`~file.read`,
       :py:meth:`~file.seek`, and :py:meth:`~file.tell` methods,
       and be opened in binary mode.
    :param mode: The mode.  If given, this argument must be "r".
    :returns: An :py:class:`~PIL.Image.Image` object.
    :exception IOError: If the file cannot be found, or the image cannot be
       opened and identified.
```

```
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/