Comments

Python Block, Inline, & Multi-Line Comments

Symbol	Description
#	***Block Comments*** start with the # sign, and then are ignored by the interpreter. ***Inline Comments*** are coded after statements to describe what they do. Comments can be used to disable statements that you do not want to run.
“ “ “ “ “ “	Three double-quote marks at the beginning and three more at the end are used to create multi-line comments. ***Multi-Line Comments*** are used to block out entire sections. This is used to disable many statements that you do not want to run.

Why use comments?

Many good programmers use comments to make their programs easier to understand as well as describe variables. Comments are great for programmers to explain code that is hard to understand, or what their code is doing and how to manipulate variables to enhance or tweak the app. It is a great way to ensure longevity of your application.

Use comments to comment out (or disable) statements that you do not want to run.

Comments are ignored by the compiler and do not impact the performance of the app.

It is always good practice to start off with basic information including your name, date, and a description of what is going on in this app.

Single Line Comments:

In [ ]:
```
#   Ernie Drumm, 9/27/2020
#  This app was created to do some really cool things.
```

Multi Line Comments:

In [11]:
```
"""
    Ernie Drumm, 9/27/2020

    This app does come really cool things.  See instructions and explainations throughout.
"""
```

In-line comments

In [ ]:
```
cars[0] = "Toyota"  # set index 0 to Toyota

cars[1] = "Audi"    # Comments at the end of the line are ignored by interperter.
```

In [12]:
```
a = 33   # single = symbol is called an assignment operator..  33 is assigned to the variable 'a'
b = 33
if b > a:
  print("b is greater than a")
elif a == b:                        # the double == symbol is used as a literal equal.
  print("a and b are equal")
```

```
a and b are equal
```

Comment out code

Sometimes you need to break a problem down into manageable parts. It is helpful to comment a few lines of code to help troubleshoot.

In [ ]:
```
ASCII_CHARS=["@","#","$","%","?","*","+",";",":",",","."]

# ASCII_CHARS=["!","@","#","$","%","^","&","?","*","+",";",":",",","."]
```

In [ ]:
```
def resize_image(image, new_width=100):
  width, height = image.size
  ratio = height / width
  new_height = int(new_width * ratio)

  resized_image = image.resize((new_width, new_height))
  # resized_image = image.resized((new_width, new_height))  spelling of "resized" vs "resize" caused the error


  return(resized_image)
```

Comments vs. NO Comments

In the example below, see if you can figure out what is going on in the first block of code? There is no comments.

In the second example, the programmer took great effort to ensure the audience understood what is going on with the code and gives many suggestions for futher learning.

In [ ]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
img = mpimg.imread('/Users/mrdru/Desktop/Python/Clipart.png')
print(img)
imgplot = plt.imshow(img)
lum_img = img[:, :, 0]
plt.imshow(lum_img)
plt.imshow(lum_img, cmap="hot")
imgplot = plt.imshow(lum_img)
imgplot.set_cmap('nipy_spectral')
imgplot = plt.imshow(lum_img)
plt.colorbar()
plt.hist(lum_img.ravel(), bins=256, range=(0.0, 1.0), fc='k', ec='k')
imgplot = plt.imshow(lum_img, clim=(0.0, 0.7))
fig = plt.figure()
ax = fig.add_subplot(1, 2, 1)
imgplot = plt.imshow(lum_img)
ax.set_title('Before')
plt.colorbar(ticks=[0.1, 0.3, 0.5, 0.7], orientation='horizontal')
ax = fig.add_subplot(1, 2, 2)
imgplot = plt.imshow(lum_img)
imgplot.set_clim(0.0, 0.7)
ax.set_title('After')
plt.colorbar(ticks=[0.1, 0.3, 0.5, 0.7], orientation='horizontal')
from PIL import Image
img = Image.open('/Users/mrdru/Desktop/Python/Clipart.png')
img.thumbnail((64, 64), Image.ANTIALIAS)  # resizes image in-place
imgplot = plt.imshow(img)
imgplot = plt.imshow(img, interpolation="nearest")
imgplot = plt.imshow(img, interpolation="bicubic")
```

Take the above code and add comments! Lots of very informative and very useful comments! Now see if the code is easier to understand?

In [ ]:
```
"""
==============
Image tutorial
==============

A short tutorial on plotting images with Matplotlib.

.. _imaging_startup:

Startup commands
===================

First, let's start IPython.  It is a most excellent enhancement to the
standard Python prompt, and it ties in especially well with
Matplotlib.  Start IPython either directly at a shell, or with the Jupyter
Notebook (where IPython as a running kernel).

With IPython started, we now need to connect to a GUI event loop.  This
tells IPython where (and how) to display plots.  To connect to a GUI
loop, execute the **%matplotlib** magic at your IPython prompt.  There's more
detail on exactly what this does at `IPython's documentation on GUI
event loops
<https://ipython.readthedocs.io/en/stable/interactive/reference.html#gui-event-loop-support>`_.

If you're using Jupyter Notebook, the same commands are available, but
people commonly use a specific argument to the %matplotlib magic:

.. sourcecode:: ipython

    In [1]: %matplotlib inline

This turns on inline plotting, where plot graphics will appear in your
notebook.  This has important implications for interactivity.  For inline plotting, commands in
cells below the cell that outputs a plot will not affect the plot.  For example,
changing the color map is not possible from cells below the cell that creates a plot.
However, for other backends, such as Qt5, that open a separate window,
cells below those that create the plot will change the plot - it is a
live object in memory.

This tutorial will use Matplotlib's imperative-style plotting
interface, pyplot.  This interface maintains global state, and is very
useful for quickly and easily experimenting with various plot
settings.  The alternative is the object-oriented interface, which is also
very powerful, and generally more suitable for large application
development.  If you'd like to learn about the object-oriented
interface, a great place to start is our :doc:`Usage guide
</tutorials/introductory/usage>`.  For now, let's get on
with the imperative-style approach:
"""

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

###############################################################################
# .. _importing_data:
#
# Importing image data into Numpy arrays
# ======================================
#
# Matplotlib relies on the Pillow_ library to load image data.
#
# .. _Pillow: https://pillow.readthedocs.io/en/latest/
#
# Here's the image we're going to play with:
#
# .. image:: ../../_static/stinkbug.png
#
# It's a 24-bit RGB PNG image (8 bits for each of R, G, B).  Depending
# on where you get your data, the other kinds of image that you'll most
# likely encounter are RGBA images, which allow for transparency, or
# single-channel grayscale (luminosity) images.  You can right click on
# it and choose "Save image as" to download it to your computer for the
# rest of this tutorial.
#
# And here we go...

img = mpimg.imread('/Users/mrdru/Desktop/Python/Clipart.png')
# my hero https://www.vhv.rs/dpng/d/418-4180695_true-form-all-might-action-figure-hd-png.png


# img = mpimg.imread('C:\Users\mrdru\Pictures\ASCII_ART\space.png')
# img = mpimg.imread('../../doc/_static/stinkbug.png')

print(img)

###############################################################################
# Note the dtype there - float32.  Matplotlib has rescaled the 8 bit
# data from each channel to floating point data between 0.0 and 1.0.  As
# a side note, the only datatype that Pillow can work with is uint8.
# Matplotlib plotting can handle float32 and uint8, but image
# reading/writing for any format other than PNG is limited to uint8
# data.  Why 8 bits? Most displays can only render 8 bits per channel
# worth of color gradation.  Why can they only render 8 bits/channel?
# Because that's about all the human eye can see.  More here (from a
# photography standpoint): `Luminous Landscape bit depth tutorial
# <https://luminous-landscape.com/bit-depth/>`_.
#
# Each inner list represents a pixel.  Here, with an RGB image, there
# are 3 values.  Since it's a black and white image, R, G, and B are all
# similar.  An RGBA (where A is alpha, or transparency), has 4 values
# per inner list, and a simple luminance image just has one value (and
# is thus only a 2-D array, not a 3-D array).  For RGB and RGBA images,
# Matplotlib supports float32 and uint8 data types.  For grayscale,
# Matplotlib supports only float32.  If your array data does not meet
# one of these descriptions, you need to rescale it.
#
# .. _plotting_data:
#
# Plotting numpy arrays as images
# ===================================
#
# So, you have your data in a numpy array (either by importing it, or by
# generating it).  Let's render it.  In Matplotlib, this is performed
# using the :func:`~matplotlib.pyplot.imshow` function.  Here we'll grab
# the plot object.  This object gives you an easy way to manipulate the
# plot from the prompt.

imgplot = plt.imshow(img)

###############################################################################
# You can also plot any numpy array.
#
# .. _Pseudocolor:
#
# Applying pseudocolor schemes to image plots
# -------------------------------------------------
#
# Pseudocolor can be a useful tool for enhancing contrast and
# visualizing your data more easily.  This is especially useful when
# making presentations of your data using projectors - their contrast is
# typically quite poor.
#
# Pseudocolor is only relevant to single-channel, grayscale, luminosity
# images.  We currently have an RGB image.  Since R, G, and B are all
# similar (see for yourself above or in your data), we can just pick one
# channel of our data:

lum_img = img[:, :, 0]

# This is array slicing.  You can read more in the `Numpy tutorial
# <https://docs.scipy.org/doc/numpy/user/quickstart.html>`_.

plt.imshow(lum_img)

###############################################################################
# Now, with a luminosity (2D, no color) image, the default colormap (aka lookup table,
# LUT), is applied.  The default is called viridis.  There are plenty of
# others to choose from.

plt.imshow(lum_img, cmap="hot")

###############################################################################
# Note that you can also change colormaps on existing plot objects using the
# :meth:`~matplotlib.cm.ScalarMappable.set_cmap` method:

imgplot = plt.imshow(lum_img)
imgplot.set_cmap('nipy_spectral')

###############################################################################
#
# .. note::
#
#    However, remember that in the Jupyter Notebook with the inline backend,
#    you can't make changes to plots that have already been rendered.  If you
#    create imgplot here in one cell, you cannot call set_cmap() on it in a later
#    cell and expect the earlier plot to change.  Make sure that you enter these
#    commands together in one cell.  plt commands will not change plots from earlier
#    cells.
#
# There are many other colormap schemes available.  See the `list and
# images of the colormaps
# <../colors/colormaps.html>`_.
#
# .. _`Color Bars`:
#
# Color scale reference
# ------------------------
#
# It's helpful to have an idea of what value a color represents.  We can
# do that by adding a color bar to your figure:

imgplot = plt.imshow(lum_img)
plt.colorbar()

###############################################################################
# .. _`Data ranges`:
#
# Examining a specific data range
# ---------------------------------
#
# Sometimes you want to enhance the contrast in your image, or expand
# the contrast in a particular region while sacrificing the detail in
# colors that don't vary much, or don't matter.  A good tool to find
# interesting regions is the histogram.  To create a histogram of our
# image data, we use the :func:`~matplotlib.pyplot.hist` function.

plt.hist(lum_img.ravel(), bins=256, range=(0.0, 1.0), fc='k', ec='k')

###############################################################################
# Most often, the "interesting" part of the image is around the peak,
# and you can get extra contrast by clipping the regions above and/or
# below the peak.  In our histogram, it looks like there's not much
# useful information in the high end (not many white things in the
# image).  Let's adjust the upper limit, so that we effectively "zoom in
# on" part of the histogram.  We do this by passing the clim argument to
# imshow.  You could also do this by calling the
# :meth:`~matplotlib.cm.ScalarMappable.set_clim` method of the image plot
# object, but make sure that you do so in the same cell as your plot
# command when working with the Jupyter Notebook - it will not change
# plots from earlier cells.
#
# You can specify the clim in the call to ``plot``.

imgplot = plt.imshow(lum_img, clim=(0.0, 0.7))

###############################################################################
# You can also specify the clim using the returned object
fig = plt.figure()
ax = fig.add_subplot(1, 2, 1)
imgplot = plt.imshow(lum_img)
ax.set_title('Before')
plt.colorbar(ticks=[0.1, 0.3, 0.5, 0.7], orientation='horizontal')
ax = fig.add_subplot(1, 2, 2)

imgplot = plt.imshow(lum_img)

imgplot.set_clim(0.0, 0.7)

ax.set_title('After')
plt.colorbar(ticks=[0.1, 0.3, 0.5, 0.7], orientation='horizontal')

###############################################################################
# .. _Interpolation:
#
# Array Interpolation schemes
# ---------------------------
#
# Interpolation calculates what the color or value of a pixel "should"
# be, according to different mathematical schemes.  One common place
# that this happens is when you resize an image.  The number of pixels
# change, but you want the same information.  Since pixels are discrete,
# there's missing space.  Interpolation is how you fill that space.
# This is why your images sometimes come out looking pixelated when you
# blow them up.  The effect is more pronounced when the difference
# between the original image and the expanded image is greater.  Let's
# take our image and shrink it.  We're effectively discarding pixels,
# only keeping a select few.  Now when we plot it, that data gets blown
# up to the size on your screen.  The old pixels aren't there anymore,
# and the computer has to draw in pixels to fill that space.
#
# We'll use the Pillow library that we used to load the image also to resize
# the image.

from PIL import Image

img = Image.open('/Users/mrdru/Desktop/Python/Clipart.png')
img.thumbnail((64, 64), Image.ANTIALIAS)  # resizes image in-place
imgplot = plt.imshow(img)

###############################################################################
# Here we have the default interpolation, bilinear, since we did not
# give :func:`~matplotlib.pyplot.imshow` any interpolation argument.
#
# Let's try some others. Here's "nearest", which does no interpolation.

imgplot = plt.imshow(img, interpolation="nearest")

###############################################################################
# and bicubic:

imgplot = plt.imshow(img, interpolation="bicubic")

###############################################################################
# Bicubic interpolation is often used when blowing up photos - people
# tend to prefer blurry over pixelated.
```

Resources

Resources used and great source for deeper learning:

1. W3Schools.com Python Comments
2. BeginnerBook Comments in Python Programming
3. Comments in Python: Best Practices video.