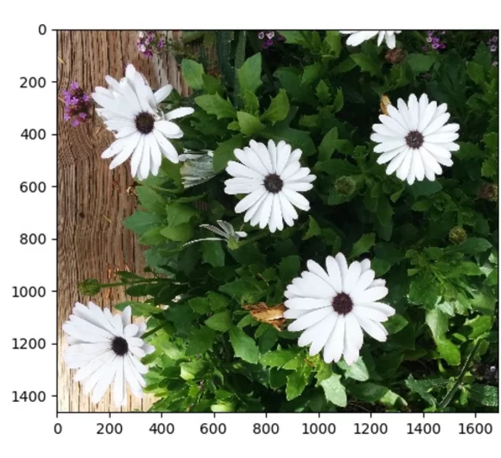
Image Processing in Python

You may already have experience with some form of image processing in your daily life. When you crop photos, apply filters on Instagram or Snapchat, or use effects like Blur or Sharpen in Photoshop, you are using algorithms that help highlight information that you want your images to convey. In scientific computing, we use image processing to make it easier to make certain measurements or to find something specific of interest within the image. Image processing is important for analyzing information from a variety of instruments, such as MRI machines, microscopes, telescopes, and of course, regular cameras.

Here are some other examples of things we might want to do:

- Identifying similarities between images (useful for tasks like Google's reverse image search)
- Make it easier to detect specific objects in an image, such as small, faint exoplanets orbiting a very bright star or a tumor in an MRI.
- Analyzing changes between images. This might be used to track how something like a bacterium is moving in a video (which is essentially a series of images), or to look for new objects that might have appeared in the sky.
- Make measurements or count objects. Perhaps you're interested in how large a galaxy is, or how many people are in an image.

Can you think of some other ways that image processing may be useful?



Take note of the picture above. Python stores images as arrays. Each pixel of the image is associated with a y-coordinate and an x-coordinate. Notice that the vertical coordinate comes first here, which is the opposite of how you're probably used to writing coordinates in math class. The origin of the plot is in the upper left corner, which is probably different from the typical xy-plane that you're used to, which has the origin in the lower left corner. Remember that Python is zero-indexed, so the first pixel is pixel 0, not pixel 1.

Understanding how to read these types of charts can be challenging. So let's try an exercise. What are the coordinates of the five centers of the petals above?

That's right, the five petals are in the following general locations:

- (350, 300)
- (1200, 200)
- (600, 800)
- (1000, 1100)
- (400, 1400)

Recall that you can read them like an (y, x) plot!

Image files

To a computer, an image is a series of numbers that are organized in a specific way. We demonstrate this in the code block below:

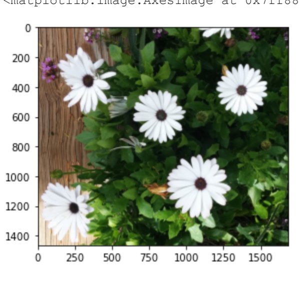
In [3]:

```
#import image processing package
from skimage import io
import matplotlib.pyplot as plt
#read in image
imagefile = io.imread('Flowers.jpg')
print("The datatype of the image is", type(imagefile))
print("The shape of the image array is", imagefile.shape)

plt.imshow(imagefile)
```

```
The datatype of the image is <class 'numpy.ndarray'>
The shape of the image array is (1464, 1687, 3)
```

Out[3]:

```
<matplotlib.image.AxesImage at 0x7ff88328c670>
```



- "Flowers.jpg" is the name of the image.
- To turn this image into a form that Python can read, we will use the free and publicly available scikit-image library (also sometimes called a package). Lesson 2 taught us that a library is a collection of code that other people can use as part of their own software projects. Note that scikit-image is imported as "skimage" because the name used inside Python is not necessarily the same as the name that we call a package in writing.
- Within the scikit-image library is a function called io.imread. When you pass your filename (which is a string) to the io.imread function, the function will return a NumPy array (notice that the scikit-image library itself uses the NumPy library). For a refresher on NumPy arrays, see Lesson 2. In order to do more with the NumPy array holding the image information, we need to store it in the variable that we choose to call "imagefile." (Note that running the code will be a little slow because the scikit-image package has to be downloaded by replit; however, if you're working with a Python installation on your own computer, you only have to download and install new packages the first time you use them).
- Since "imagefile" is a NumPy array, we can print out its shape, which is (1464, 1687, 3). 1464 represents the number of pixels in the image in the vertical direction. 1687 represents the number of pixels in the horizontal direction. Finally, 3 represents the number of colors that we can break this image down into: red, green, and blue (in that order). It is very common for computers to represent colors as some combination of these three colors. This is known as RGB for short. Each of the three values should be an integer from 0 to 255. The larger the number, the more the individual color contributes to the overall color. If all three values are 0, the pixel is black. If they are all 255, the pixel is white.

We can see what the array looks like with the statement "print(imagefile)." Feel free to try it above! It will print with ellipses because the array is so large, so this isn't a good strategy for looking up what values correspond to what pixels in the image.

However, since an image in Python is just a NumPy array, we can use the same functions that we would for other NumPy arrays. For example, we can select part of the image by using normal indexing. We can figure out the color of the pixel at a vertical position of 1200 and a horizontal position of 1491with the command "imagefile[1200,1491]."

Can you determine the color of the pixel at (1200,1000)?

In [7]:

```
print("The color of the pixel at a vertical position of 1200 and a horizontal positio
print(imagefile[1200,1000])
```

```
The color of the pixel at a vertical position of 1200 and a horizontal position of 100
0 is:
[241 246 250]
```

If you google this color and use an RGB color convertor you will see it's a white-ish gray

Image features

Oftentimes, the types of images you'll analyze for a scientific purpose won't be in color. This might be because color is not important to the task at hand (e.g., if you're tracking the motion of something under a microscope) or because the image is mapping some other quantity entirely (like an MRI, which maps signals coming from your tissues after a magnetic field is applied). Thus, image pixels do not necessarily have RGB values associated with each pixel. Instead, each pixel will be associated with a single value that denotes the amount of signal being measured (i.e., grayscale). In the case of photographs, the signal is light.

We will read the flowers image into Python again, but this time we will convert to grayscale. The code is nearly identical to what was shown previously, but we add an "as_gray=True" argument to the imread function call.

Note that when we print the shape of the image now, we get (1464, 1687) instead of (1464, 1687,3) because each pixel now corresponds to a single intensity value rather than an RGB color. The value is also normalized so that the maximum value corresponds to 1, since the computer does not have information about your units.

You should also now get a plot of the image in grayscale like the one below:

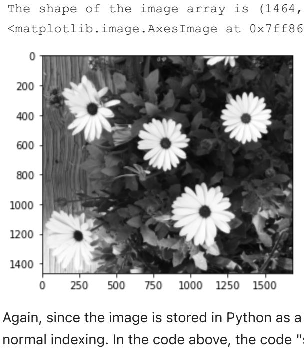
In [17]:

```
#import image processing package
from skimage import io
import matplotlib.pyplot as plt
#read in image
imagefile = io.imread('Flowers.jpg', as_gray = True)
print("The shape of the image array is", imagefile.shape)

plt.imshow(imagefile, cmap = 'gray')
```

```
The shape of the image array is (1464, 1687)
```

Out[17]:

```
<matplotlib.image.AxesImage at 0x7ff86157a8b0>
```



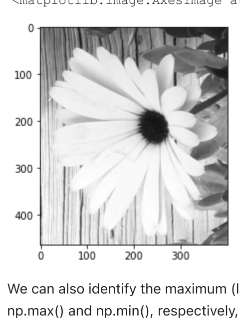
Again, since the image is stored in Python as a NumPy array, we can select part of the image by using normal indexing. In the code above, the code "subimage = imagefile[1000:, :400]" will select all the pixels with y-positions above 1000 and x-positions up to 400 (i.e., the lower-left corner of the image. This is like cropping an image. The result is shown below. Note that the coordinates are now re-numbered so that (0,0) corresponds to the origin of the subimage.

In [10]:

```
subimage = imagefile[1000:, :400]
plt.imshow(subimage, cmap = 'gray')
```

Out[10]:

```
<matplotlib.image.AxesImage at 0x7ff86e070ca0>
```



We can also identify the maximum (largest) and minimum (smallest) values of the image using np.max() and np.min(), respectively, each of which takes an array as an argument. We see that the maximum and minimum values are 1.0 and 0.00196, respectively. In this particular case, 1.0 corresponds to completely white, 0.0 corresponds to completely back, and values in-between correspond to shades of gray (larger values are lighter).

In [12]:

```
import numpy as np
#use NumPy to get basic information about the image
maxvalue = np.max(imagefile)
minvalue = np.min(imagefile)
print("The maximum value of the image is", maxvalue)
print("The minimum value of the image is", minvalue)
max_yvals, max_xvals = np.where(imagefile==maxvalue)
min_yvals, min_xvals = np.where(imagefile==minvalue)
```

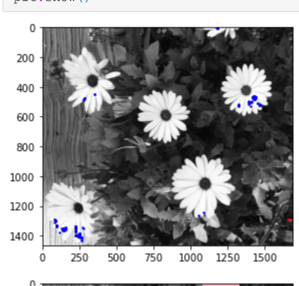
```
The maximum value of the image is 1.0
The minimum value of the image is 0.001964313725490196
```

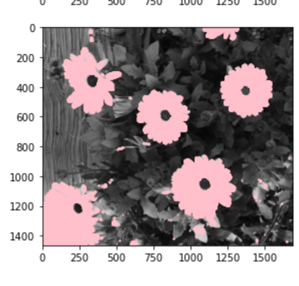
What is more useful is finding out where in the image these maximum and minimum values occur using the np.where() function, which returns all the values that meet a certain condition. For example, the command "max_yvals, max_xvals = np.where(imagefile==maxvalue)" will return the y and x coordinates of all the pixels in the image that are equal to the maximum value of the image. The locations of the pixels matching the minimum value can be found in a similar way.

Using matplotlib, we take our grayscale image and make a scatterplot of the locations of the brightest pixels (in blue) and the darkest pixels (in red). We see that the maximum pixel values can be found in the flowers, while the darkest pixels (see the red dot in the lower right corner) can be found in a shadow.

In [13]:

```
#Figure out where minima and maxima are
f3 = plt.figure()
plt.ylim(ymin = 1464, ymax = 0)
plt.xlim(xmin = 0, xmax = 1687)
plt.imshow(imagefile, cmap = 'gray')
plt.scatter(max_xvals, max_yvals, s = 2, color = 'blue')
plt.scatter(min_xvals, min_yvals, s = 5, color = 'red')
plt.show()

upper_yvals, upper_xvals = np.where(imagefile>=0.9*maxvalue)
f4 = plt.figure()
plt.ylim(ymin = 1464, ymax = 0)
plt.xlim(xmin = 0, xmax = 1687)
plt.imshow(imagefile, cmap = 'gray')
plt.scatter(upper_xvals, upper_yvals, s = 1, color = 'pink')
plt.show()
```





Checkpoint

You will be applying what you've learned to an image of holiday cookies ("cookies.jpg")

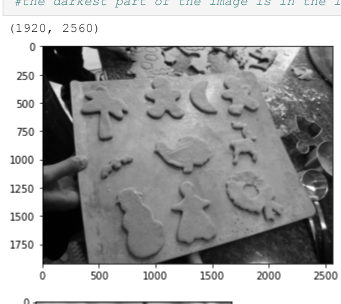
- Find the dimensions (shape) of the image using NumPy
- Use array slicing to select just the part of the image that shows the snowman cookie
- Use NumPy to identify where the maximum and minimum values of the image occur. What kind of features do they correspond to?

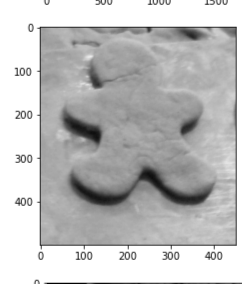
Solution

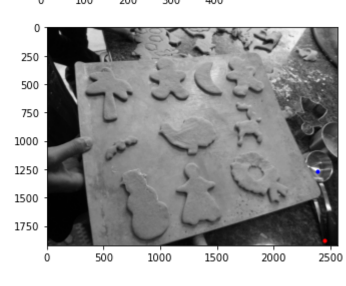
In [14]:

```
#import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from skimage import io
#read in image
imagefile = io.imread('cookies.jpg', as_gray = True)
#Show the image using matplotlib
fig1 = plt.figure()
plt.imshow(imagefile, cmap = 'gray')
plt.savefig("cookies_grayscale.jpg")
#Find the shape of the image using NumPy
print(imagefile.shape)
#Use array slicing to select the part of the image that shows the snowman cookie, and
subimage=imagefile[240:740, 850:1300]
#your solution does not have to have the exact same indices, but they should be relat
fig2 = plt.figure()
plt.imshow(subimage, cmap = 'gray')
plt.savefig("snowman.jpg")
#use NumPy to identify where the maximum and minimum values of the image occur. What k
maxvalue = np.max(imagefile)
minvalue = np.min(imagefile)
max_yvals, max_xvals = np.where(imagefile==maxvalue)
min_yvals, min_xvals = np.where(imagefile==minvalue)
#Figure out where minima and maxima are
f3 = plt.figure()
plt.ylim(ymin = 1920, ymax = 0)
plt.xlim(xmin = 0, xmax = 2560)
plt.imshow(imagefile, cmap = 'gray')
plt.scatter(max_xvals, max_yvals, s = 10, color = 'blue')
plt.scatter(min_xvals, min_yvals, s = 10, color = 'red')
plt.savefig("locations2.jpg")
#the brightest part of the image comes from the part of the metal measuring cup on the
#the darkest part of the image is in the lower right corner, in a shadow
```

```
(1920, 2560)
```







Thresholding

Python libraries like scikit-image offer some advanced image processing techniques used by scientists. We'll touch on a few below. If you're interested in learning more details, take a look at the code and tutorials in http://scikit-image.org/docs/dev/auto_examples.

We briefly touched upon the idea of thresholding in the previous section when we picked out all the pixels that were above a certain value in order to identify which parts of the image corresponded to the white flowers. In general, thresholding is useful for picking out certain objects if these objects have a similar color (or intensity) to one another and are different in intensity from other objects in the picture. The threshold we set in the last section, 90%, was a good guess that let us identify the flowers, but how do we know whether there are better values we could have picked?

One popular technique is Otsu's method, which analyzes the values of all the pixels in a specific image to choose the best threshold value to separate different objects. The scikit-image library has a function called "threshold_otsu" that uses Otsu's method to calculate a threshold. Because this is a very famous thresholding technique, Otsu's method is included in a lot of image processing software.

We find that the threshold value of the flowers picture is 0.55 (recall that the maximum value is 1.0). We create a thresholded image with the command "thresholded_image = imagefile > threshold", which returns a 2D boolean array to determine which parts of the image are above or below the threshold. A boolean can be either True or False. If the value of a pixel is True, the pixel intensity is above the threshold (lighter than the threshold). If the value of the pixel is False, the pixel intensity is below the threshold (darker than the threshold).

In computer science, True and False are usually set equivalent to 1 and 0, respectively. Recall that in our image, 1 corresponded to white and 0 corresponded to black. So "thresholded_image" is a NumPy array that represents a binary image, where every pixel can only take on one of two possible values. Again, we can plot this array/image using matplotlib. Note that this method enables us to pick out the regions of the image with the flower petals, but you can't really see the leaves anymore. This is useful if you want to simplify the image so that you can focus on the flowers.

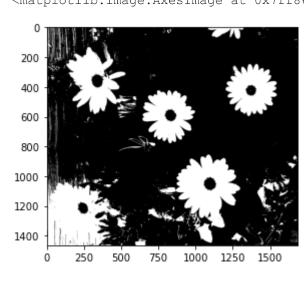
In [16]:

```
import matplotlib.pyplot as plt
from skimage import io
from skimage.filters import threshold_otsu

#read in image
imagefile = io.imread("Flowers.jpg", as_gray = True)
threshold = threshold_otsu(imagefile)
print("The threshold value returned by Otsu's method is %.2f" % threshold)
thresholded_image = imagefile > threshold

plt.imshow(thresholded_image, cmap = 'gray')
```

```
The threshold value returned by Otsu's method is 0.55
```

Out[16]:

```
<matplotlib.image.AxesImage at 0x7ff860b126a0>
```



Edge Detection

In the previous section, we picked out objects of interest (i.e., the flowers) by focusing on the brightest regions, since we know that the flowers are lighter than the other parts of the image. Another way you can pick out an object of interest from image is through edge detection. The basic idea behind edge detection methods is that they check how quickly intensity values change across an image. This assumes that within a single object, the intensity does not change too much, but the intensities change a lot between objects. For example, within a single flower petal, there are small variations in color, but the pixels are mostly similar in intensity to one another. But when you cross over to a leaf, the pixels suddenly become much darker.

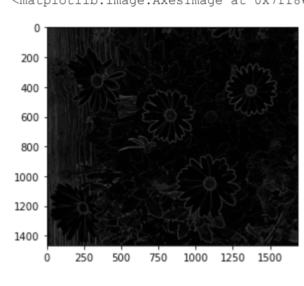
One widely-used edge detection method is called the Roberts algorithm, which is also included as a function in the scikit-image library. We apply this algorithm to the same image of flowers we analyzed in the previous section, and we see that the processed image mostly consists of the edges of the flowers, with some contribution from the wooden frame on the side and the leaves on the bottom of the image.

In [19]:

```
#import image processing package
from skimage import io
from skimage.filters import roberts
#import plotting package
import matplotlib.pyplot as plt
#read in image
imagefile = io.imread('Flowers.jpg', as_gray = True)
#find object edges
edges = roberts(imagefile)
plt.imshow(edges, cmap = "gray")
```

Out[19]:

```
<matplotlib.image.AxesImage at 0x7ff8615d9d90>
```



Template Matching

Sometimes, you'll want to figure out where or how many times a certain kind of object appears in an image. (Imagine, for instance, that you're dealing with a huge number of images that would be difficult for a single person to look through quickly). This is where template matching comes in. For example, if you're curious about how many times the white flower shows up in the image, you can start with a template, and then make a map of where the image of interest best matches the template. For our template image (which we refer to as "flowertemplate.jpg"), we are using one of the flowers from the image and assuming that it is similar enough to the other flowers in the image that computer software will identify the other flowers as a good match to the template image, without confusing other objects as flowers.

We do not necessarily have to use part of the original image as a template. You could just as easily select a flower from a different image. Sometimes people will simulate an image to use as a template based on their knowledge of what the object should look like.

In [21]:

```
import numpy as np
#import image processing package
from skimage import io
from skimage.feature import match_template
#import plotting package
import matplotlib.pyplot as plt
#read in image
imagefile = io.imread('Flowers.jpg', as_gray = True)
#read in template image
template = io.imread('flowertemplate.jpg', as_gray = True)
#calculate how well the template matches different locations in the image
match_intensity = match_template(imagefile, template)
```

Underneath the hood, the function is essentially figuring out where the array of numbers represented by the flower template best matches a sub-section of the array of numbers representing the image you're processing. This function returns a 2D array of numbers (which we call "match_intensity" in the code below) indicating how well a location in the image matches to the template. Higher numbers mean the match is better.

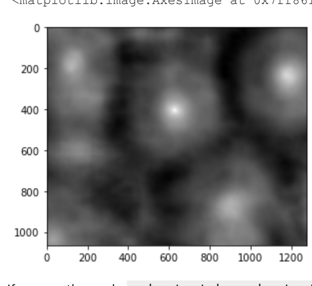
We can make a plot of this 2D array. We can see that the brightest spot in the map is near the center, which corresponds to the location of the flower that was originally used to make the template. But notice that we also get four other bright spots, corresponding to the other four similar flowers that are also in the image. We can locate these other four flowers using our old friends np.max() and np.where() to find the local maxima in the image (i.e., the pixels in the image that are brighter than anything nearby).

In [22]:

```
plt.imshow(match_intensity, cmap = 'gray')
```

Out[22]:

```
<matplotlib.image.AxesImage at 0x7ff861f2a5e0>
```



If we run the code `y_bestmatch, x_bestmatch = np.where(np.max(match_intensity)==match_intensity)`, then we'll only get the location of the brightest spot at the center. To get the location of a different local maximum, we need to select only part of the array using array slicing: `y_match, x_match = np.where(np.max(match_intensity[:400, 1000:])==match_intensity[:400,1000:])`

This line of code will help you find the location of the flower in the upper right corner of the intensity map. However, the value that you get for x_match will be 176, because the code only looks at the part of the array where the x-value is larger than 1000. So, to get back the value of x that corresponds to the original match_intensity plot, you need to add 1000 so that you get 1176.

In [24]:

```
#find out where the best match occurs
y_bestmatch, x_bestmatch = np.where(np.max(match_intensity)==match_intensity)
print("The best match occurs at an x value of %d and a y value of %d" % (x_bestmatch[
#find out where match occurs to flower in upper right corner:
y_match, x_match = np.where(np.max(match_intensity[:400, 1000:])==match_intensity[:40
print("A match also occurs at an x value of %d and a y value of %d" % (x_match[0]+100
```

```
The best match occurs at an x value of 625 and a y value of 405
A match also occurs at an x value of 1176 and a y value of 237
```

Summary

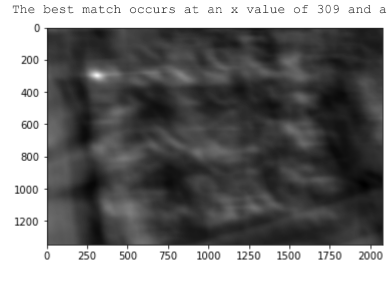
You will identify where a palm tree cookie template ("cookietemplate.jpg") best matches a tray of holiday cookies ("cookies.jpg").

Solution

In [26]:

```
#import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from skimage import io
from skimage.feature import match_template
#read in image
imagefile = io.imread('cookies.jpg', as_gray = True)
#read in template image
template = io.imread('cookietemplate.jpg', as_gray = True)
#calculate how well the template matches different locations in the image
cookiematch_intensity = match_template(imagefile, template)
f1 = plt.figure()
plt.imshow(cookiematch_intensity, cmap = 'gray')
plt.savefig("cookiematch_intensity.jpg")
#find out where the best match occurs
y_bestmatch, x_bestmatch = np.where(np.max(cookiematch_intensity)==cookiematch_intens
print("The best match occurs at an x value of %d and a y value of %d" % (x_bestmatch[
```

```
The best match occurs at an x value of 309 and a y value of 299
```



Wrap-up

You should now understand the following:

- Some of the roles that image processing plays in scientific research
- How images are stored in Python
- How to use Python libraries to extract useful information from images

In [ ]: