The Little Known OGrid Function in Numpy

...and how to use it to easily transform images

Ritvik Kharkar Follow
Dec 20, 2019 · 6 min read ★

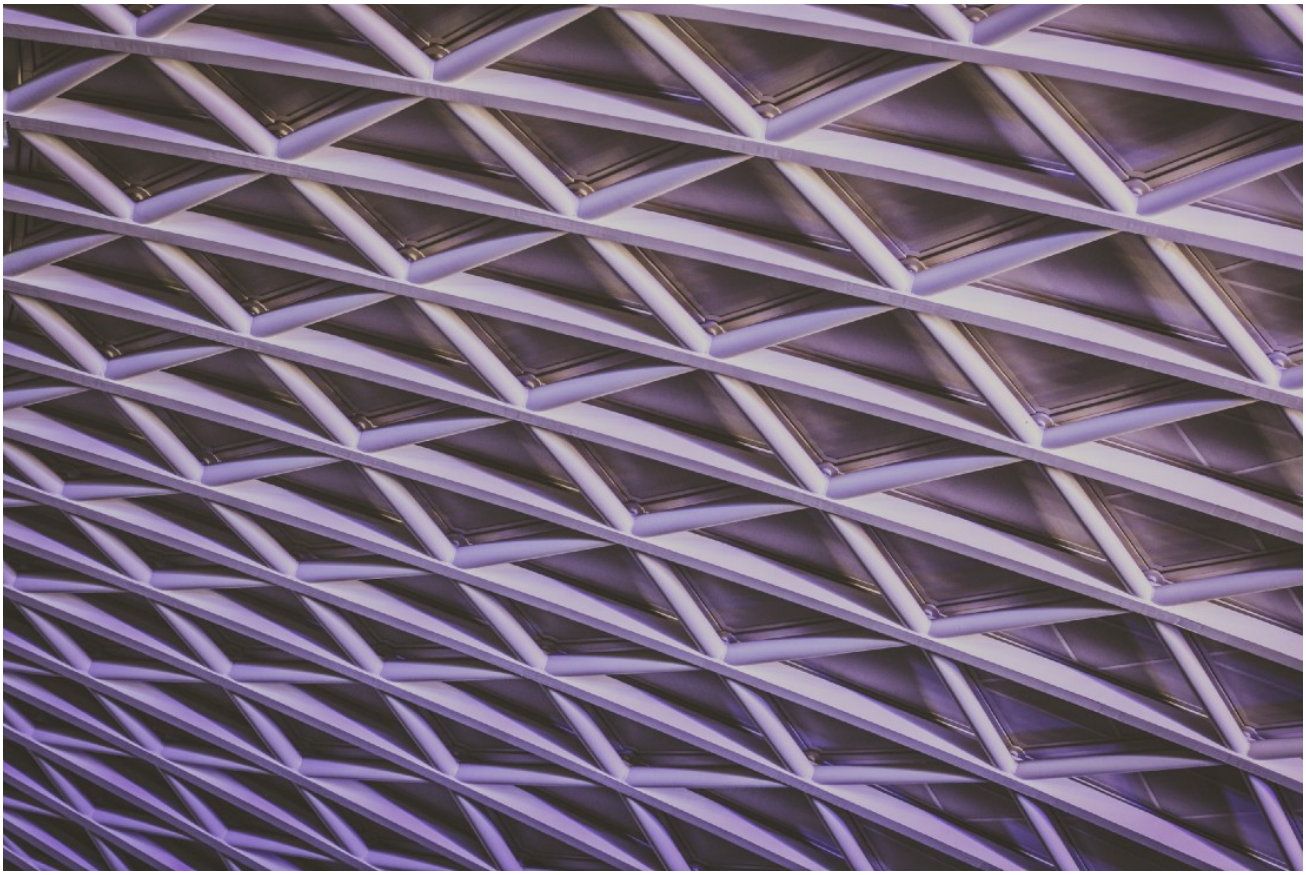


Photo by Zara Walker on Unsplash

This past quarter, I was helping my students learn the powerful numerical processing library in Python known as **Numpy**. For a good chunk of the course, we were using

Numpy to perform transformations on images, because images are, after all, just big arrays of numbers.

Soon enough, we started talking about how to transform an image, from basic transformations like making an image black & white to more complex edits like adding a halo around an image. It was during this discussion that I learned about the little known ***ogrid*** function in Numpy and how we can leverage it to more easily transform images.

. . .

What is ogrid?

Ogrid stands for "**open grid**" and basically provides a way to act on specific pixels of an image based on their row and column index. To start from basics, here is how to call the ogrid function in Numpy assuming you've imported Numpy as np.

```
In [196]: #constructing an open grid (ogrid) of size 10 by 5
          x,y = np.ogrid[0:10, 0:5]

In [197]: x

Out[197]: array([[0],
                 [1],
                 [2],
                 [3],
                 [4],
                 [5],
                 [6],
                 [7],
                 [8],
                 [9]])

In [198]: y

Out[198]: array([[0, 1, 2, 3, 4]])
```

We see here that ogrid takes in a range of the x-coordinate (0 through 10, not including 10) and a range of the y-coordinate (0 through 5, not including 5). We store the results of ogrid in two variables, **x and y**.

Showing the contents of x reveals that it is a list of size-one lists, each of which is a number between 0 and 9. Showing the contents of y reveals that it is a list of a single list, which is just the numbers between 0 and 4.

. . .

How does ogrid help us?

Well, suppose you have a matrix that is 10 by 5, just like the dimensions of the ogrid. We see below that indexing the matrix using the ogrid gives back *exactly* the matrix itself.

```
In [24]: m
Out[24]: array([[0.07749084, 0.87956259, 0.10474302, 0.85719911, 0.5044291 ],
                [0.21683803, 0.03723761, 0.440383  , 0.13695166, 0.26043965],
                [0.52762569, 0.15152548, 0.03382974, 0.40416059, 0.76873017],
                [0.37052596, 0.16215551, 0.50901557, 0.9488231 , 0.64763655],
                [0.67453808, 0.61504895, 0.47001945, 0.28792319, 0.66392799],
                [0.75156029, 0.23164353, 0.96732614, 0.27135384, 0.45421792],
                [0.0897914 , 0.03247285, 0.02920534, 0.1593243 , 0.17521615],
                [0.78633486, 0.87023167, 0.37723074, 0.2748243 , 0.81170206],
                [0.80010869, 0.822923  , 0.35548626, 0.84422796, 0.09693641],
                [0.60004223, 0.63321734, 0.01625481, 0.44351901, 0.92787423]])
```

```
In [25]: m[x,y]
Out[25]: array([[0.07749084, 0.87956259, 0.10474302, 0.85719911, 0.5044291 ],
                [0.21683803, 0.03723761, 0.440383  , 0.13695166, 0.26043965],
                [0.52762569, 0.15152548, 0.03382974, 0.40416059, 0.76873017],
                [0.37052596, 0.16215551, 0.50901557, 0.9488231 , 0.64763655],
                [0.67453808, 0.61504895, 0.47001945, 0.28792319, 0.66392799],
                [0.75156029, 0.23164353, 0.96732614, 0.27135384, 0.45421792],
                [0.0897914 , 0.03247285, 0.02920534, 0.1593243 , 0.17521615],
                [0.78633486, 0.87023167, 0.37723074, 0.2748243 , 0.81170206],
                [0.80010869, 0.822923  , 0.35548626, 0.84422796, 0.09693641],
                [0.60004223, 0.63321734, 0.01625481, 0.44351901, 0.92787423]])
```

More importantly, we can use the x and y returned by the ogrid to create something

called a **"mask"**, isolating only the matrix elements we care about, based on their row and column index.

```
In [28]: mask = (x > 5)|(y < 3)
         mask

Out[28]: array([[ True,  True,  True, False, False],
                [ True,  True,  True, False, False],
                [ True,  True,  True, False, False],
                [ True,  True,  True, False, False],
                [ True,  True,  True, False, False],
                [ True,  True,  True, False, False],
                [ True,  True,  True,  True,  True],
                [ True,  True,  True,  True,  True],
                [ True,  True,  True,  True,  True],
                [ True,  True,  True,  True,  True]])
```

In the code above, note that we only care about matrix elements whose row index (x) is bigger than 5 or whose column index (y) is less than 3. The result is a mask, which is an **array of True/False** indicating where the condition holds and where it does not.

Why is this helpful?

Because now, we can use this mask to act on *only* the elements of our matrix where the mask is True (or equivalently, False).

```
In [29]: m[mask] = 0
         m

Out[29]: array([[0.        , 0.        , 0.        , 0.85719911, 0.5044291 ],
                [0.        , 0.        , 0.        , 0.13695166, 0.26043965],
                [0.        , 0.        , 0.        , 0.40416059, 0.76873017],
                [0.        , 0.        , 0.        , 0.9488231 , 0.64763655],
                [0.        , 0.        , 0.        , 0.28792319, 0.66392799],
                [0.        , 0.        , 0.        , 0.27135384, 0.45421792],
                [0.        , 0.        , 0.        , 0.        , 0.        ],
                [0.        , 0.        , 0.        , 0.        , 0.        ],
                [0.        , 0.        , 0.        , 0.        , 0.        ],
                [0.        , 0.        , 0.        , 0.        , 0.        ]])
```

Above, we set any matrix element to 0 where the mask is True, giving the resulting matrix. Apply the same train of thought to an entire image and we see that we can use ogrid to create masks that help us act on *specific pixels* of our image. Let's see a few examples!

. . .

Reading the Image

Let's first read in the image we'll be working with.

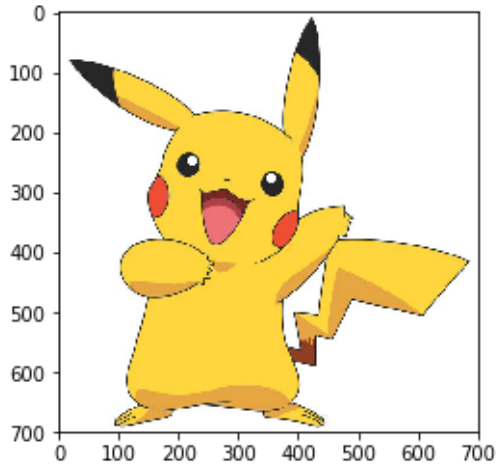
```
import numpy as np
import matplotlib.pyplot as plt

#read the image
img = plt.imread('mouse.jpg')

#show the image
plt.imshow(img)
```

read_mouse_image.py hosted with ❤️ by **GitHub** **view raw**

The result is our favorite electric mouse.



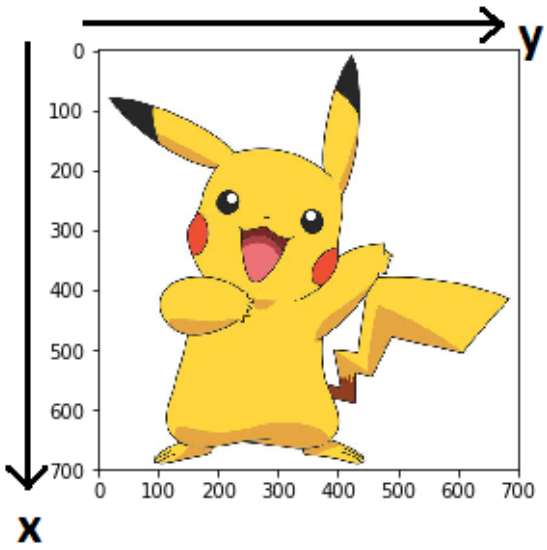
Then, let's generate the ogrid for this image.

```
#get the dimensions of the image
n,m,d = img.shape

#create an open grid for our image
x,y = np.ogrid[0:n, 0:m]
```

ogrid_for_pikachu.py hosted with ❤️ by **GitHub** **view raw**

One important note is the direction of the axes in Numpy. Contrary to how we are used to seeing x and y axes, the axes in a Numpy image are as below:



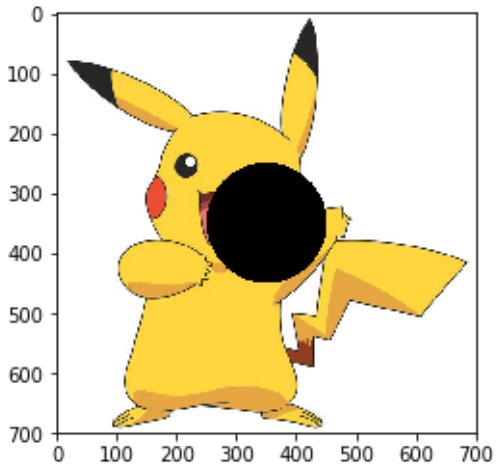
Adding Geometric Shapes

One of the coolest and most instructive image transformations with ogrid involves adding geometric shapes to your image. Since most basic geometric shapes have simple mathematical formulas, we can perform functions on the x and y returned by ogrid to recreate these formulas. For example, let's see some code to create a black circle in the middle of our image.

```
1  #operate on a copy of the image
2  copyImg = img.copy()
3
4  #get the x and y center points of our image
5  center_x = n/2
6  center_y = m/2
7
8  #create a circle mask which is centered in the middle of the image, and with radius 100
9  circle_mask = (x-center_x)**2 + (y-center_y)**2 <= 100**2
10
11 #black out anywhere within the circle mask
12 copyImg[circle_mask] = [0,0,0]
13
14 #show the image
15 plt.imshow(copyImg)
```

circle_mask.py hosted with ❤ by **GitHub** **view raw**

The key part of this code is on line 9, where you'll recognize the formula for a circle from your Algebra class. This particular circle is centered at the middle of our image, and has radius 100 pixels. We then use this circle mask to black out any pixels in the image that are included in the mask, resulting in the image below.



Nice! What if we want to make a square? Well, its arguably even easier.

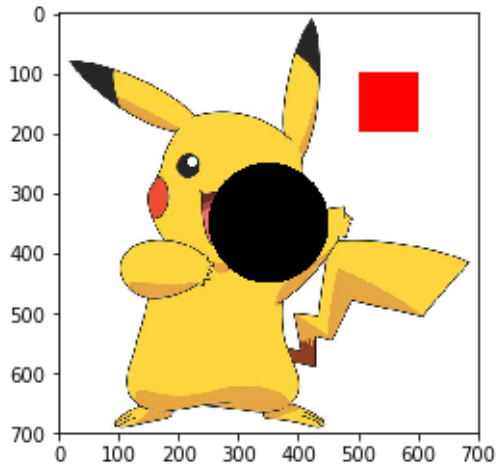
```
#create a square mask around the top right of the image
square_mask = (x<200)&(x>100)&(y<600)&(y>500)

#make the square red
copyImg[square_mask] = [255,0,0]

#show image
plt.imshow(copyImg)
```

square_mask.py hosted with ❤ by **GitHub** **view raw**

We just construct the square (or rectangle) as a composition of four filters on the x and y returned by ogrid. We then set to red any pixel in the image included in the square mask.

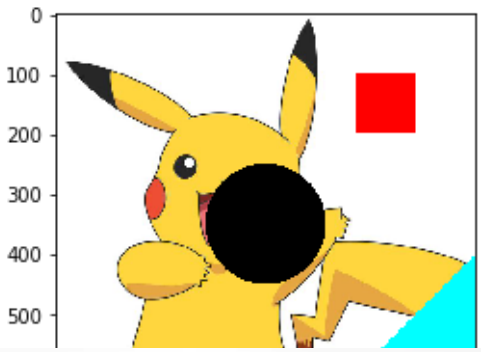


Finally, what if we want a cyan colored triangle in the bottom right corner of our image? Well, we can work out the mathematical formula, and code it as follows.

Giving the following result.

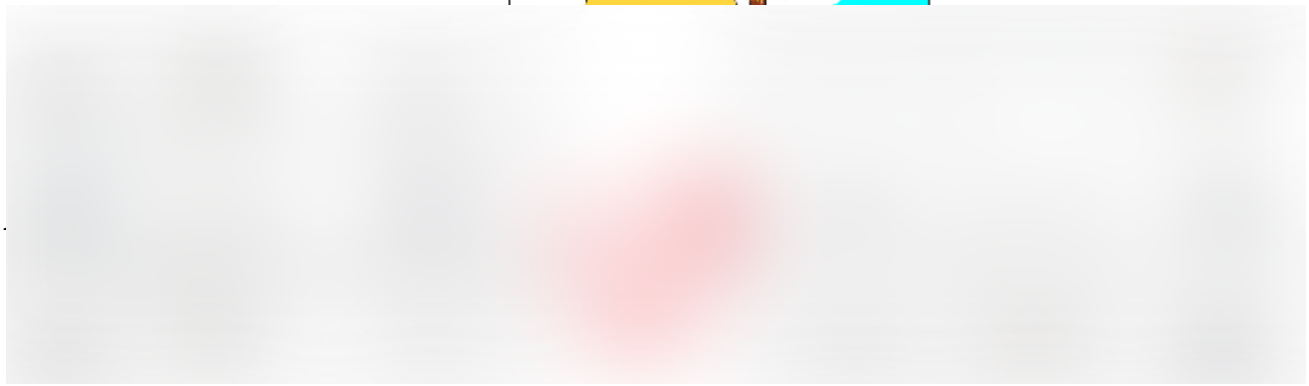
. . .

90 Degree Rotation



Next, let's see how we might u egrees. In fact, we won't write much code here, b nderstand the concept.

If we take a simple 3x3 image ockwise, we get the following result.



Mathematically, this is like taking the pixel that lives at indexes **(i,j)** and mapping it to a new location **(j,-i)**. Applying that transformation to every pixel in the image might seem daunting and seems like we need to write a few for loops, but using ogrid, the task becomes a one-liner.

Namely, if our image is called ***img***, then indexing the image as ***img[y,-x]***, using the x and y returned by ogrid, we get exactly the rotated image.

. . .

Creating a Halo

To wrap up, let's look at a slightly more complex image transformation. Namely, we want to create a **halo** around the image.

Specifically, we want to add some positive values to all the pixels of our image, and the further a pixel is from the center, the bigger the value we will add. In total, this will have the effect of "washing out" pixels near the corners and edges and have little effect on pixels near the center of the image. The fully commented code is below.

There is a good amount of code here and I encourage you to read it and try it for yourself. In a nutshell we:

- Get the squared distance of each pixel from the center
- Normalize these squared distances
- Add the normalized squared distances to each pixel in the image. This way, pixels further from the center get big values added and pixels near the center get very small values added
- The total effect is a "halo" on our image

. . .

And that's all! Hopefully, you learned a bit about the very useful ogrid functions in Numpy.

Best of luck! ~
