Numpy Array Indexing

Thomas Schwarz, SJ

Numpy Array Generation Synthesis

- In practice:
    - We will use Pandas data frames when getting data from the web
    - We will numpy arrays in order to speed up calculations
- So, we concentrate on what we need for the latter task

NumPy Array Attributes

- The number of dimensions: ndim
- The values of the dimensions as a tuple: shape
- The size (number of elements)

```
>>> tensor
array([[[2.11208424, 2.01510638, 2.03126777, 1.89670846],
        [1.94359036, 2.02299445, 2.08515919, 2.05402626],
        [1.8853457 , 2.01236192, 2.07019962, 1.93713157]],

       [[1.84275427, 1.99537922, 1.96060154, 1.90020305],
        [2.00270166, 2.11286224, 2.03144254, 2.06924855],
        [1.95375653, 2.0612986 , 1.82571628, 1.86067971]]])
>>> tensor.ndim
3
>>> tensor.shape
(2, 3, 4)
>>> tensor.size
24
```

NumPy Array Attributes

- The data type: dtype
  - can be bool, int, int64, uint, uint64, float, float64, complex …
    - Easier to use than its sounds
    - This is why Numpy can be so fast
- The size of a single element in bytes: itemsize
- The size of the total array: nbytes

NumPy Array Indexing

- How to access / modify elements:
- Single elements
  - Use the bracket notation [ ]
    - Single array: Same as in standard python

```
>>> vector = np.random.normal(10,1,(5))
>>> print(vector)
[10.25056641 11.37079651 10.44719557 10.54447875 10.43634562]
>>> vector[4]
10.436345621654919
>>> vector[-2]
10.544478746079845
```

NumPy Arrays Indexing

- Matrix and tensor elements:
  - Shortcut: a single bracket and a comma separated tuple

```
>>> tensor
array([[[2.11208424, 2.01510638, 2.03126777, 1.89670846],
        [1.94359036, 2.02299445, 2.08515919, 2.05402626],
        [1.8853457 , 2.01236192, 2.07019962, 1.93713157]],

       [[1.84275427, 1.99537922, 1.96060154, 1.90020305],
        [2.00270166, 2.11286224, 2.03144254, 2.06924855],
        [1.95375653, 2.0612986 , 1.82571628, 1.86067971]]])
>>> tensor[0,0,1]
2.015106376191313
```

NumPy Arrays Indexing

- Multiple bracket notation
  - We can also use the Python indexing of multi-dimensional lists using several brackets

```
>>> tensor[0][1][2]
2.085159191502853
```

  - It is more writing and more error prone than the single bracket version

NumPy Arrays Indexing

- We can also define slices

```
>>> vector = np.random.normal(10,1,(3))
>>> vector
array([10.61948855,  7.99635252,  9.05538706])
>>> vector[1:3]
array([7.99635252, 9.05538706])
```

NumPy Arrays Indexing

- In Python, slices are new lists
- In NumPy, slices are **not** copies
  - Changing a slice changes the original
    - Based on usage pattern
    - Avoiding unnecessary copies makes Numpy fast.

NumPy Arrays Indexing

- Example:
  - Create an array

```
>>> vector = np.random.normal(10,1,(3))
>>> vector
array([10.61948855,  7.99635252,  9.05538706])
```

  - Define a slice

```
>>> x = vector[1:3]
```

NumPy Arrays Indexing

- Example (cont.)
  - Change the first element in the slice

```
>>> x[0] = 5.0
```

  - Verify that the change has happened

```
>>> x
array([5.        , 9.05538706])
```

  - But the original has also changed:

```
>>> vector
array([10.61948855,  5.        ,  9.05538706])
```

NumPy Arrays Indexing

- Slicing does **not** makes copies
  - This is done in order to be efficient
    - Numerical calculations with a large amount of data get slowed down by unnecessary copies

NumPy Arrays Indexing

- If we want a copy, we need to make one with the copy method
- Example:
    - Make an array

```
>>> vector = np.random.randint(0,10,5)
>>> vector
array([0, 9, 5, 7, 8])
```

    - Make a copy of the array

```
>>> my_vector_copy = vector.copy()
```

NumPy Arrays Indexing

- Example (continued)
  - Change the middle elements in the copy

```
>>> my_vector_copy[1:-2]=100
```

  - Check the change

```
>>> my_vector_copy
array([  0, 100, 100,   7,   8])
```

  - Check the original

```
>>> vector
array([0, 9, 5, 7, 8])
```

  - No change!

NumPy Arrays Indexing

- Multi-dimensional slicing
    - Combines the slicing operation for each dimension

```
>>> slice = tensor[1:, :2, :1]
>>> slice
array([[[1.84275427],
        [2.00270166]]])
```

NumPy Arrays Indexing

- Multi-dimensional slicing
    - Use : in the dimensions where you do not want to slice

```
A = np.random.normal(10, 1, (3,4,5))
A[:,2:4, 1:2]
```

```
array([[[ 9.30306142],
        [10.84579805]],

       [[ 8.54188872],
        [10.78481198]],

       [[ 9.62540173],
        [10.70995867]]])
```

NumPy Arrays Conditional Selection

- We can create an array of Boolean values using comparisons on the array

```
>>> array = np.random.randint(0,10,8)
>>> array
array([2, 4, 4, 0, 0, 4, 8, 4])
>>> bool_array = array > 5
>>> bool_array
array([False, False, False, False, False,
       False,  True, False])
```

NumPy Arrays Conditional Selection

- We can then use the Boolean array to create a selection from the original array

```
>>> selection=array[bool_array]
>>> selection
array([8])
```

- The new array only has one element!

Selftest

- Can you do this in one step?
  - Create a random array of 10 elements between 0 and 10
  - Then select the ones larger than 5

Selftest Solution

- Solution:
  - Looks a bit cryptic
    - First, we create an array

```
>>> arr = np.random.randint(0,10,10)
>>> arr
array([3, 2, 7, 8, 7, 2, 1, 0, 4, 8])
```

    - Then we select in a single step

```
>>> sel = arr[arr>5]
>>> sel
array([7, 8, 7, 8])
```

NumPy Arrays Conditional Selection

- Let's try this out with a matrix
  - We create a vector, then use **reshape** to make the array into a vector
    - Recall: the number of elements needs to be the same

```
>>> mat = np.arange(1,13).reshape(3,4)
>>> mat
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

NumPy Arrays Conditional Selection

- Now let's select:

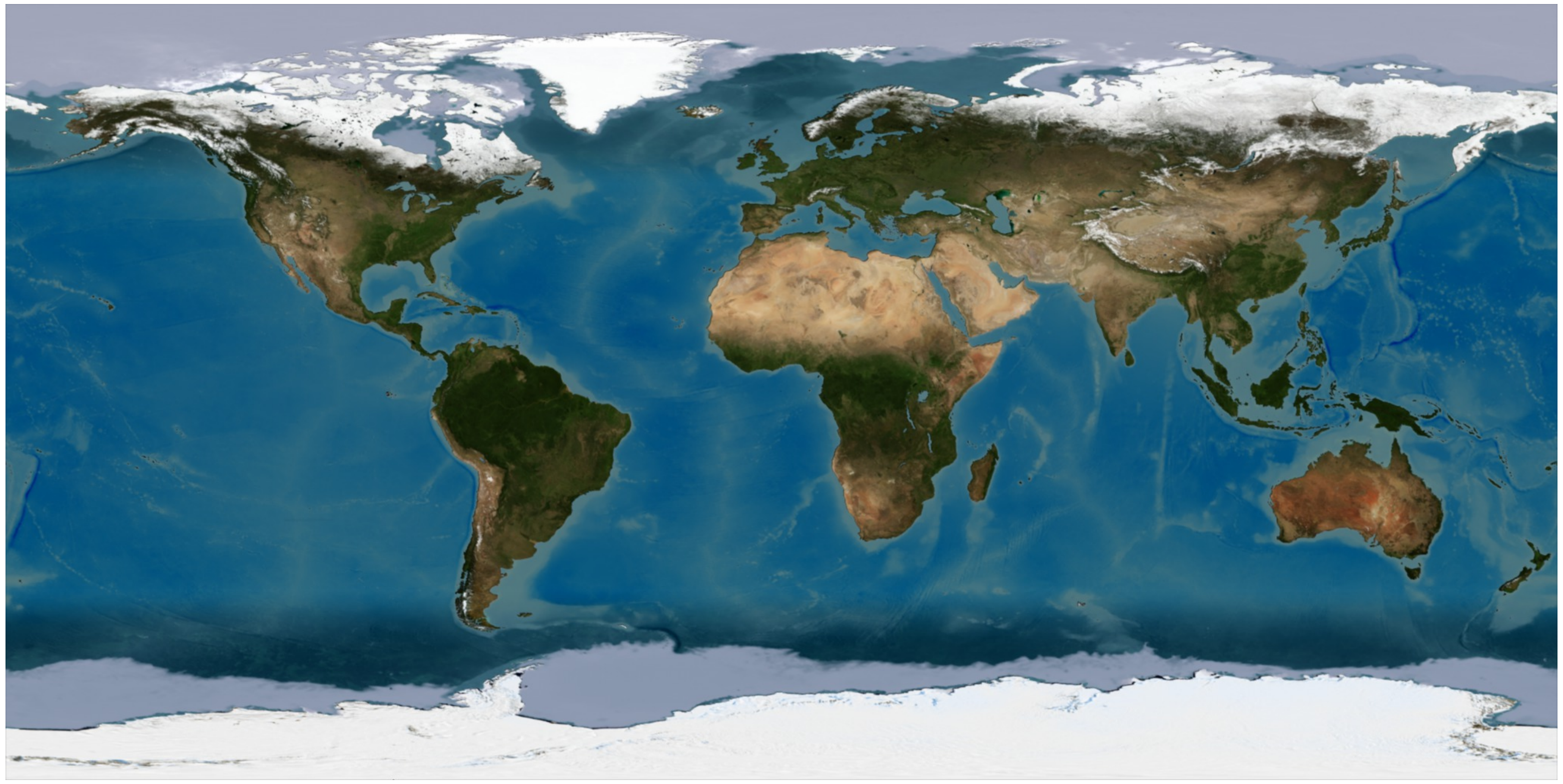
```
>>> mat1 = mat[mat>6]
>>> mat1
array([ 7,  8,  9, 10, 11, 12])
```

- This is no longer a matrix, which makes sense:
  - We remove elements, so we would have a matrix with holes

Slicing

- Photo Manipulation
  - Need to install imageio and matplotlib

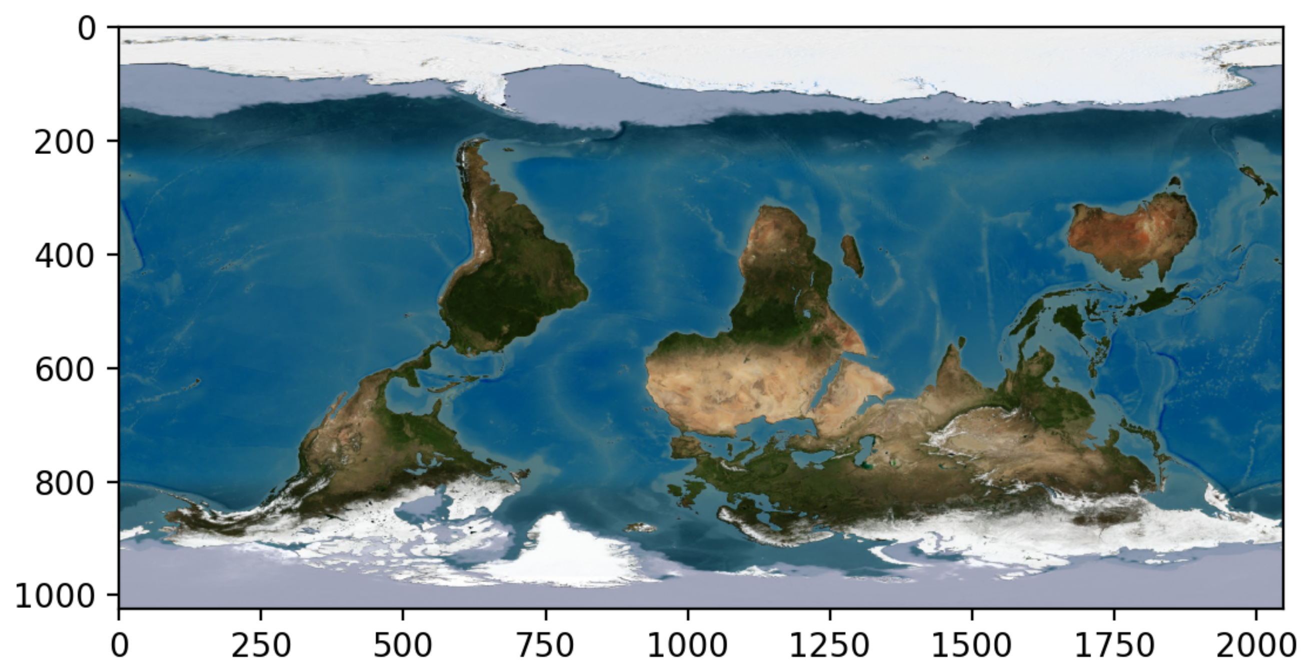
```
import imageio
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

  - Get a photo as a 3-dimensional array
    - ```
      im = mpimg.imread('earth.jpg')
      print(im.shape)
      ```
      ```

Slicing

- Display the photo
- ```
  plt.imshow(im)
  plt.show()
  ```


  ```

Slicing

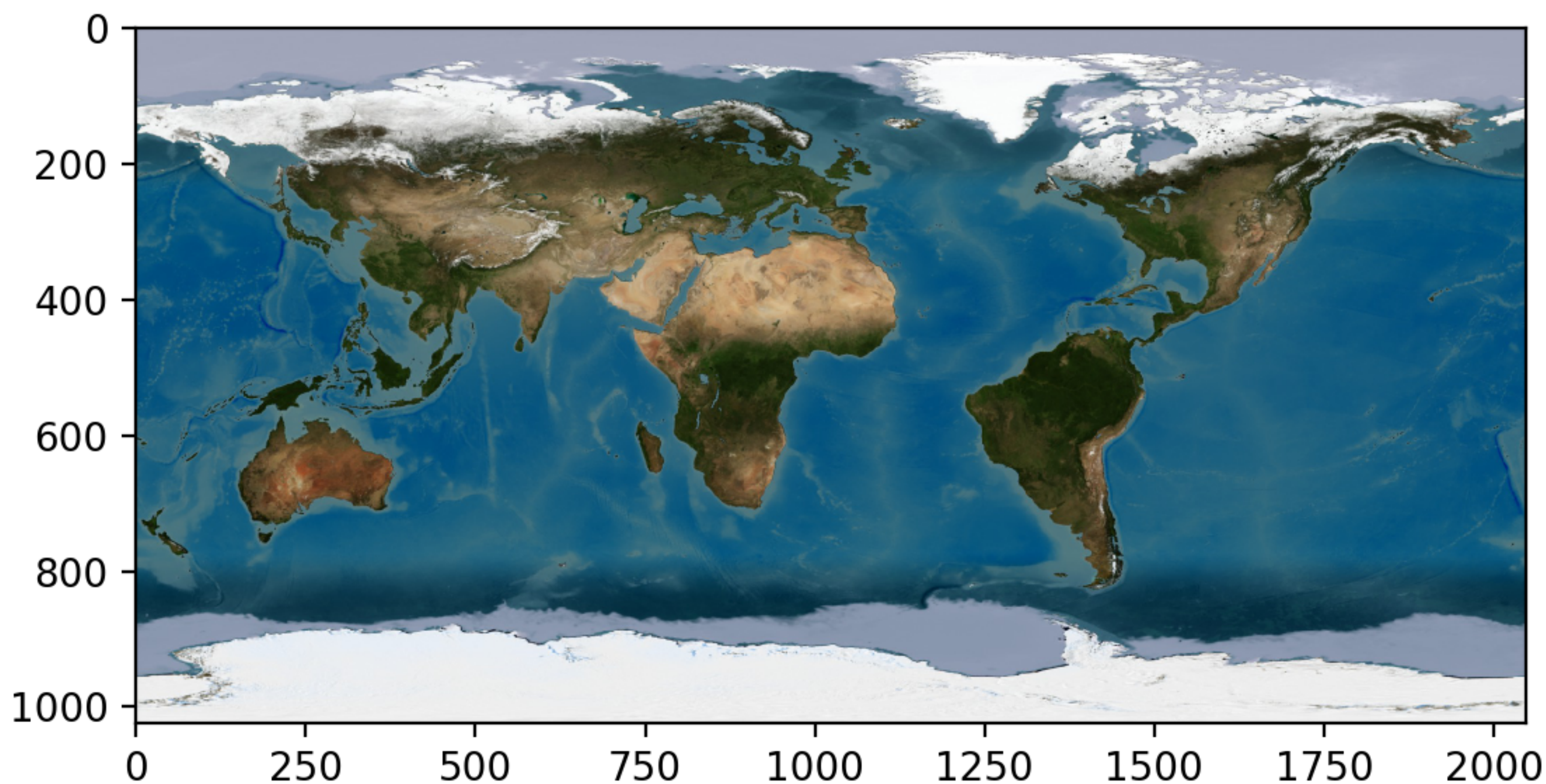
- Swap first coordinate

```
plt.imshow(im[::-1,])
plt.show()
```



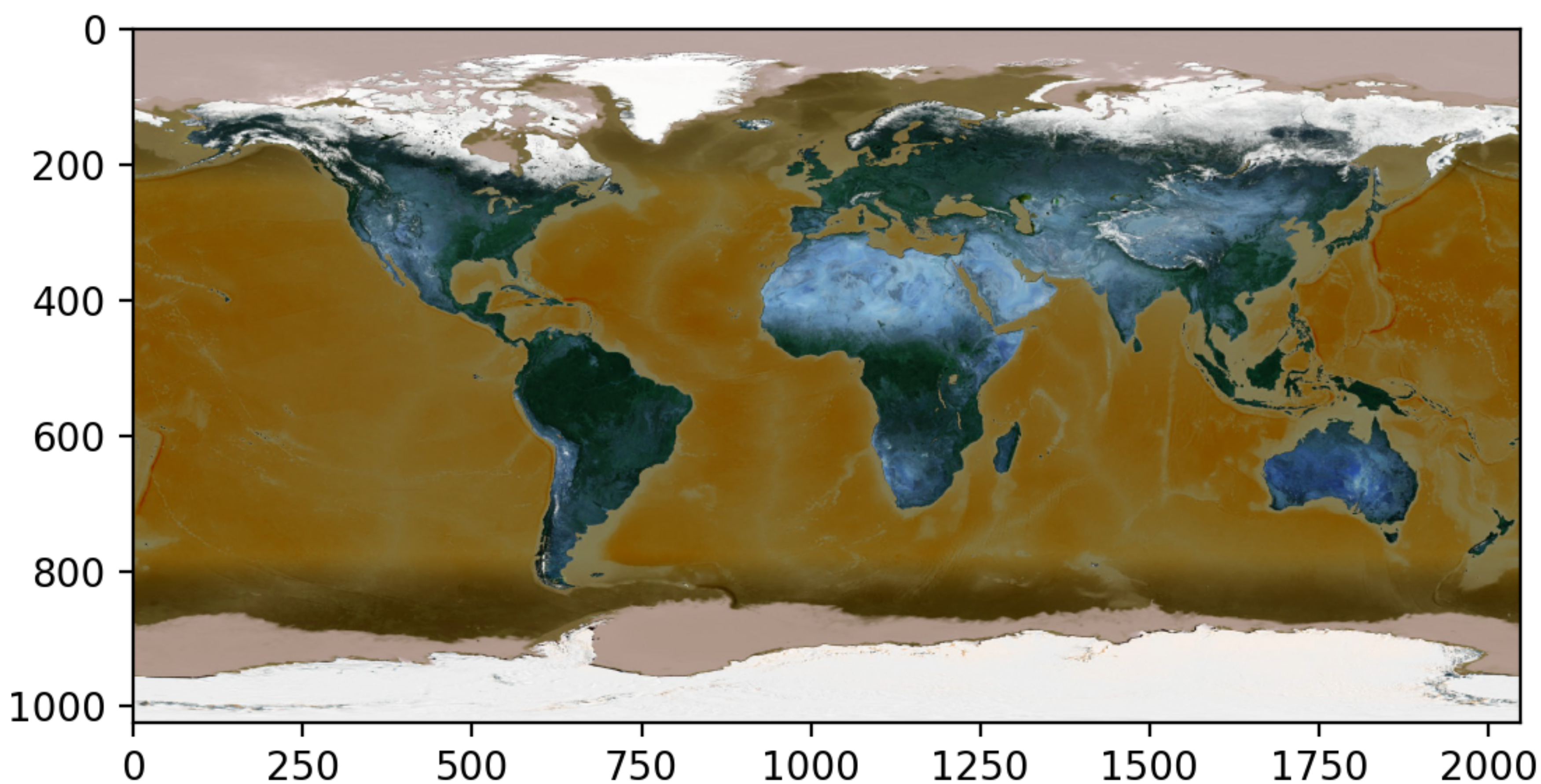
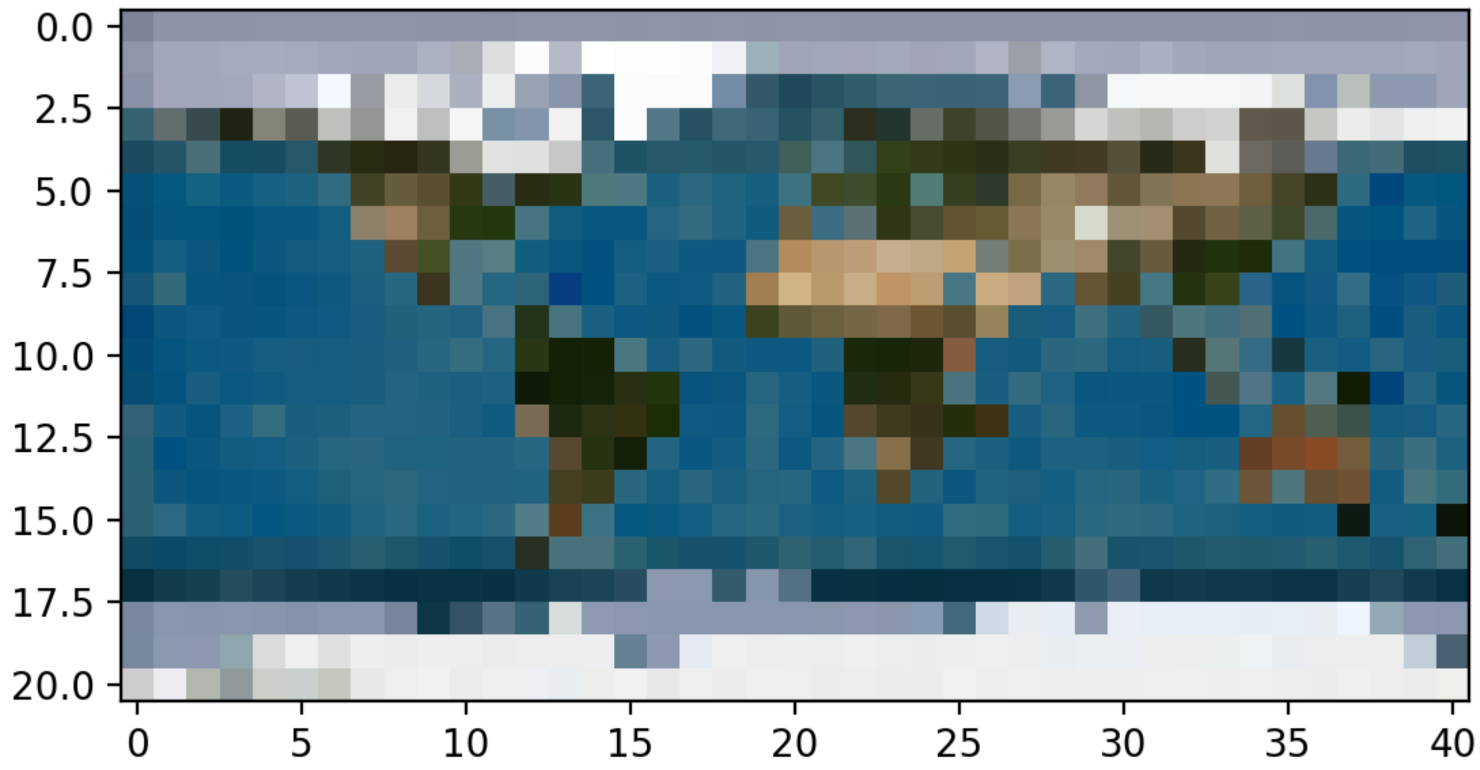
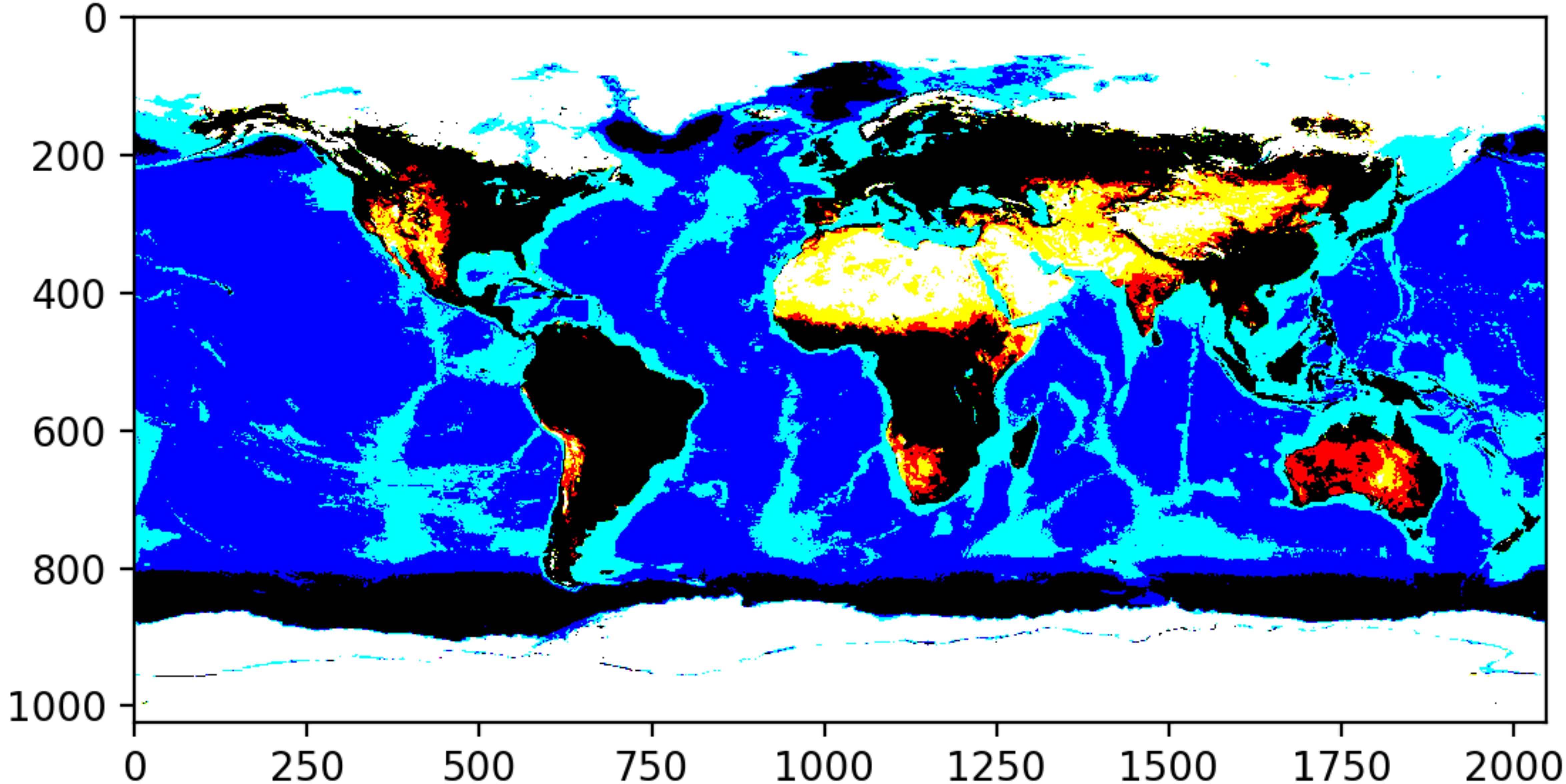
Slicing

- Swap second coordinate
-
```
plt.imshow(im[:,::-1,])
plt.show()
```

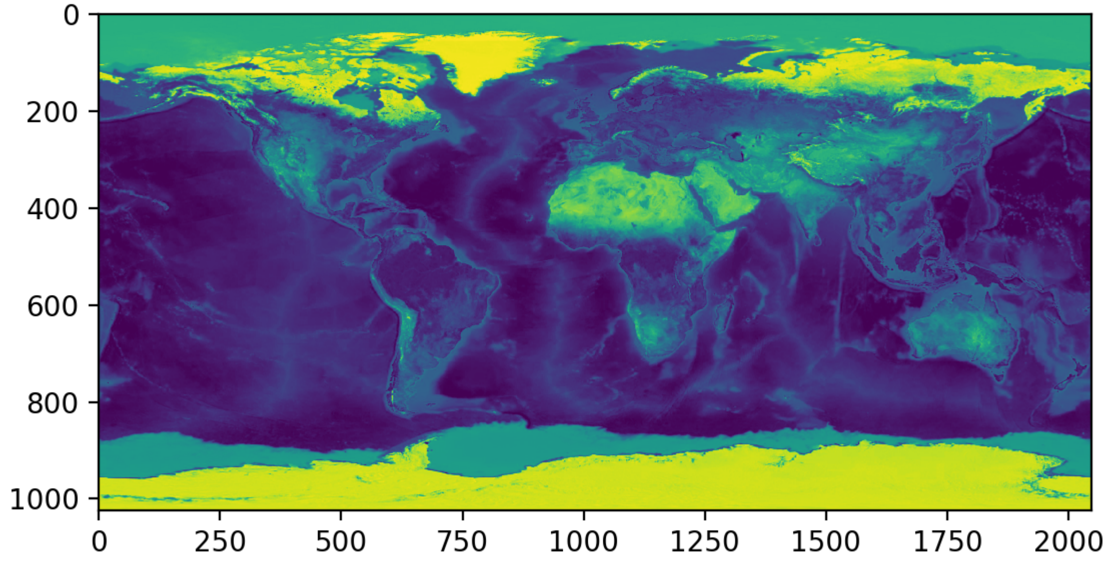
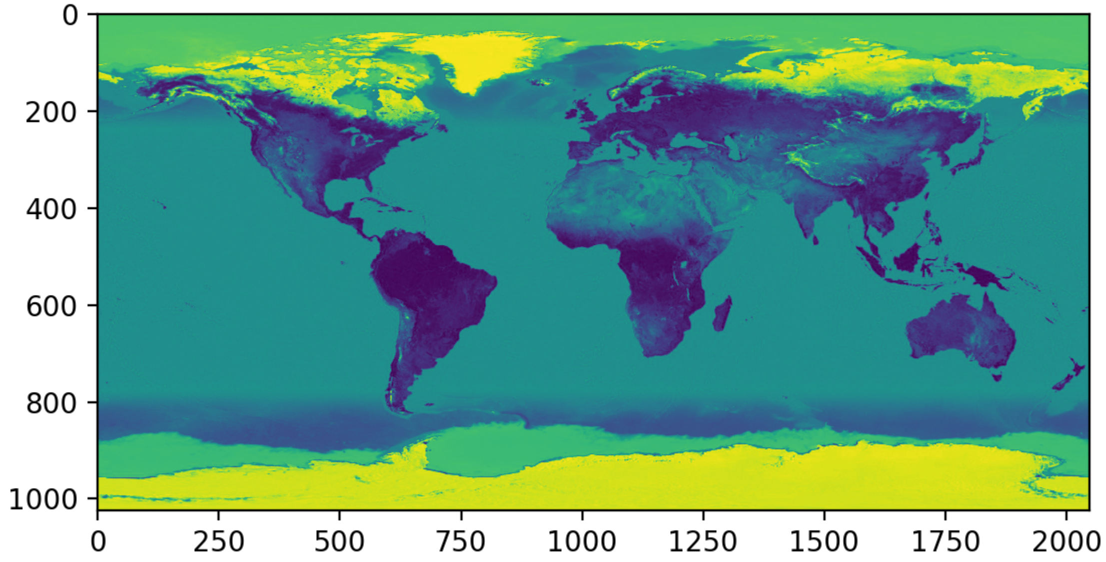
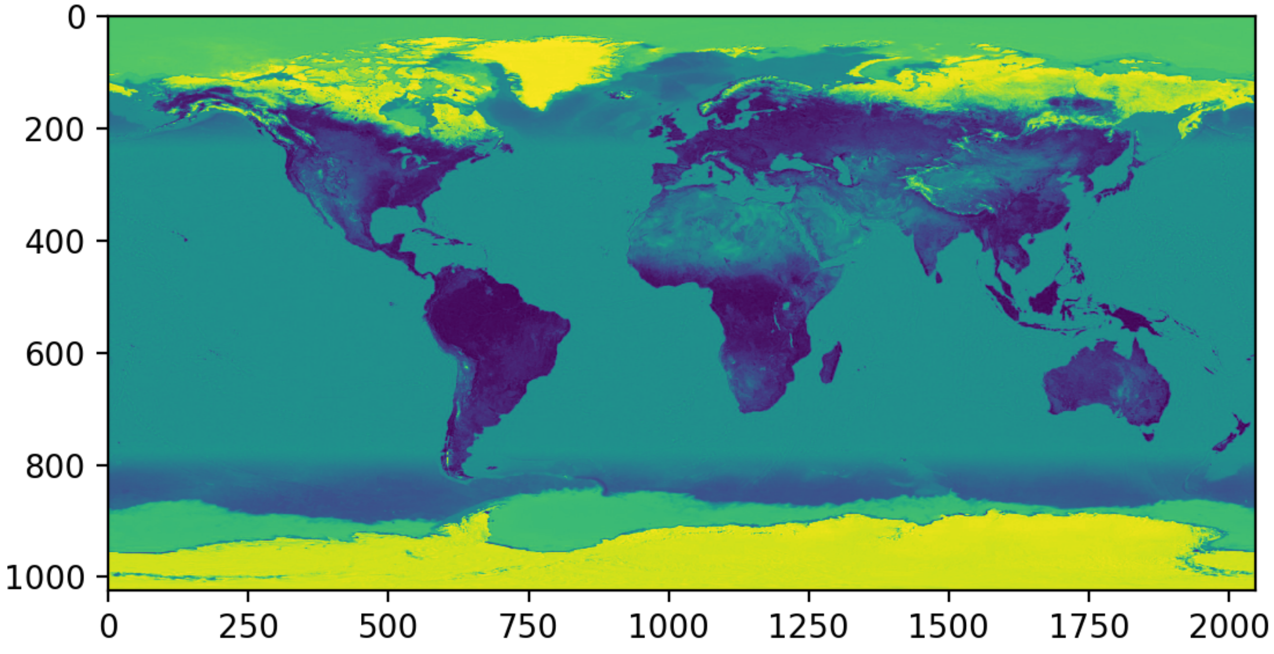
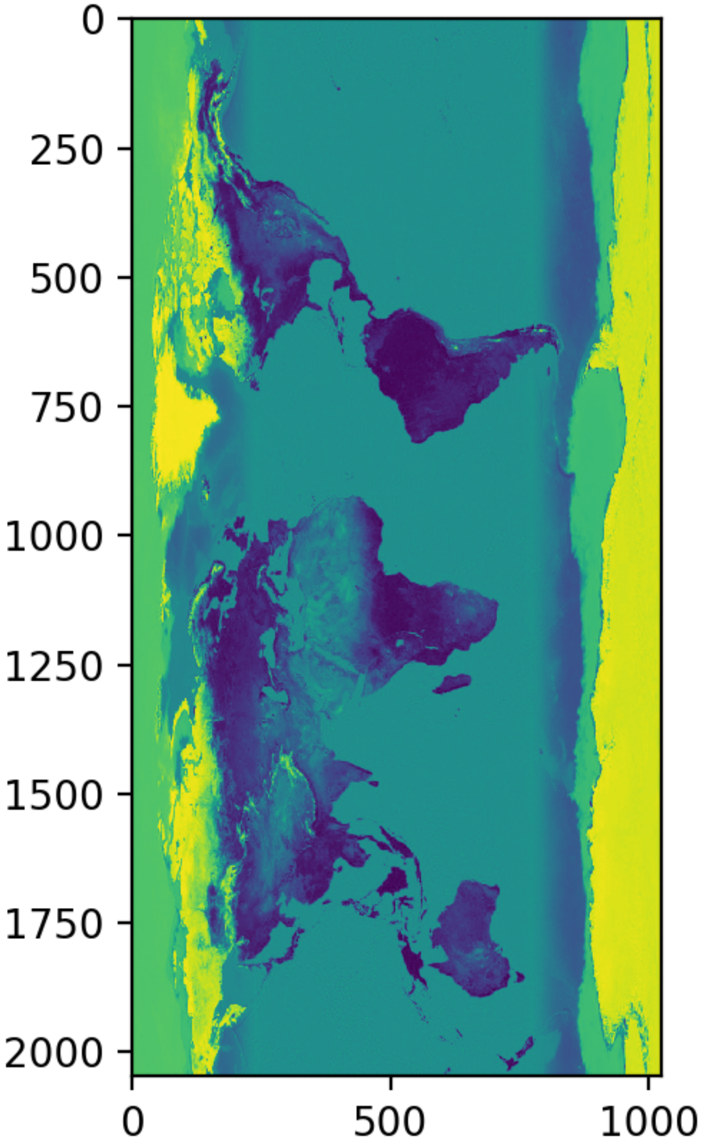
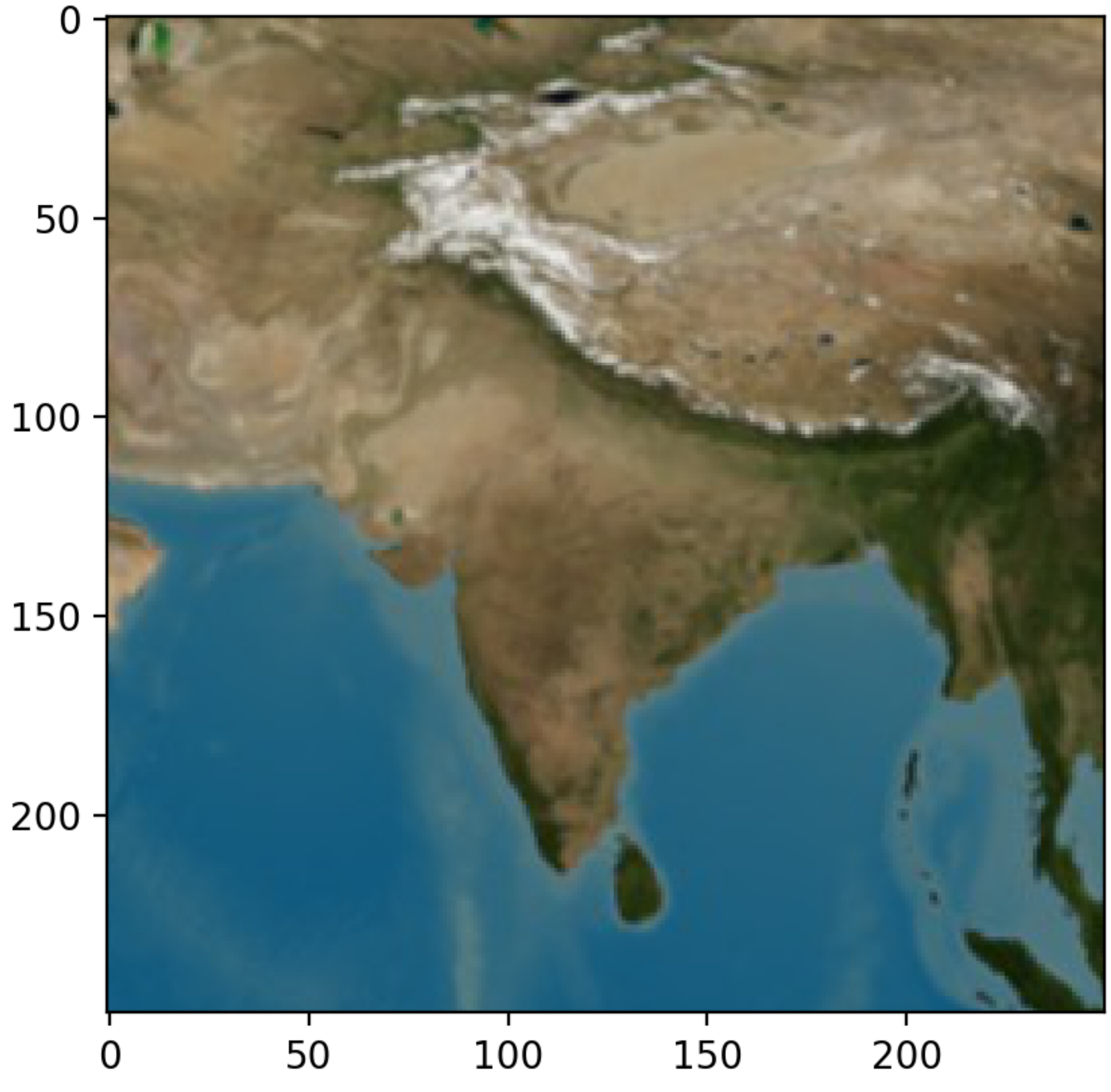


Slicing

- Swap third coordinate (color coordinate)
  - ```
    plt.imshow(im[:,:,::-1])
    plt.show()
    ```


    ```

Slicing

- Take every 50th line and column

```
plt.imshow(im[::50, ::50,])
plt.show()
```



Slicing

- We can also apply functions
  - np.where allows us to replace values
    - `image = np.where(im>100, 255, 0)`
      - Where-ever the value of the image is less than 100, replace it with 0
      - Otherwise, replace it with 255

Slicing



Slicing

- Can use a sub-image

```
plt.imshow(im[:,:,0])
plt.show()

plt.imshow(im[:,:,1])
plt.show()

plt.imshow(im[:,:,2])
plt.show()
```

Slicing


0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
2000


0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
2000


0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
2000

Slicing

- Can use the transpose

```
plt.imshow(im[:,:,2].T)
plt.show()
```



Slicing

- Or just concentrate on the essential

```
plt.imshow(im[250:500,1350:1600, : ])
```

Slicing



NumPy Operations

- Numpy allows fast operations on array elements
- We can simply add, subtract, multiply or divide by a scalar

```
>>> vector = np.arange(20).reshape(4,5)
>>> vector
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> vector += 1
>>> vector
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20]])
```

NumPy Operations

- Numpy also allows operations between arrays

```
>>> mat = np.random.normal(0,1,(4,5))
>>> mat
array([[ 0.04646031, -1.32970787,  1.16764921, -0.48342653,  0.42295389],
       [ 0.70547825,  1.51980589,  1.46902433, -0.46742839,  1.42472386],
       [ 0.78756679, -0.39975927,  1.24411043, -0.67336526, -0.92416835],
       [ 0.4708628 , -0.29419976, -0.58634161,  0.29038393, -0.78814955]])
>>> vector + mat
array([[ 1.04646031,  0.67029213,  4.16764921,  3.51657347,  5.42295389],
       [ 6.70547825,  8.51980589,  9.46902433,  8.53257161, 11.42472386],
       [11.78756679, 11.60024073, 14.24411043, 13.32663474, 14.07583165],
       [16.4708628 , 16.70580024, 17.41365839, 19.29038393, 19.21185045]])
```

NumPy Operations

- What happens if there is an error?
  - Python would throw an exception, but not so NumPy
    - Example: Create two vectors, one with a zero

```
>>> vector = np.arange(5)
>>> vector2 = np.arange(2,7)
```

- If we divide, we get a warning
- But the result exists, with an inf value for infinity

```
>>> vec = vector2/vector
Warning (from warnings module):
  File "<pyshell#11>", line 1
RuntimeWarning: divide by zero encountered in true_divide
>>> vec
array([       inf, 3.        , 2.        , 1.66666667, 1.5       ])
```

NumPy Operations

- If we divide 0 by 0, we get an nan -- not a value

```
>>> vec=np.arange(4)
>>> vec
array([0, 1, 2, 3])
>>> vec/vec

Warning (from warnings module):
  File "<pyshell#15>", line 1
RuntimeWarning: invalid value encountered in
true_divide
array([nan,  1.,  1.,  1.])
```

NumPy Operations

- There are rules for how to define operations with nan and inf, that make intuitive sense
  - IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754)
- We can create inf directly by saying `np.inf`
  - Example: Infinity divided by infinity is not defined

```
>>> np.inf/np.inf
nan
```

Operations between Vectors and Matrices

- Adding two vectors:

```
>>> v1 = np.array([1,2,3])
>>> v2 = np.array([5,4,3])
>>> v1 + v2
array([6, 6, 6])
```

Operations between Vectors and Matrices

- Adding two matrices

```
>>> m1 = np.array([[1,2,3],[4,5,6],[9,10,0]])
>>> m1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 9, 10,  0]])
>>> m2 = np.array([[4,2,0],[7,3,1],[5,1,2]])
>>> m2
array([[4, 2, 0],
       [7, 3, 1],
       [5, 1, 2]])
>>> m1+m2
array([[ 5,  4,  3],
       [11,  8,  7],
       [14, 11,  2]])
```

Operations between Vectors and Matrices

- Scalar multiplication

```
>>> v = np.array([5,3,-2,4])
>>> 5*v
array([ 25,  15, -10,  20])
```

Operations between Vectors and Matrices

- Scalar multiplication

```
>>> m1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 9, 10,  0]])

>>> 3*m1
array([[ 3,  6,  9],
       [12, 15, 18],
       [27, 30,  0]])
```

Operations between Vectors and Matrices

- Element-wise multiplication **is not matrix multiplication**

```
>>> m1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 9, 10,  0]])
>>> m2
array([[4, 2, 0],
       [7, 3, 1],
       [5, 1, 2]])
>>> m1*m2
array([[ 4,  4,  0],
       [28, 15,  6],
       [45, 10,  0]])
```

Operations between Vectors and Matrices

- **Matrix multiplication uses the (new) @ operator**
  - Python 3.5 and later

```
>>> m1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 9, 10,  0]])
>>> m2
array([[4, 2, 0],
       [7, 3, 1],
       [5, 1, 2]])
>>> m1@m2
array([[ 33,  11,   8],
       [ 81,  29,  17],
       [106,  48,  10]])
```

Operations between Vectors and Matrices

- Can be used to multiply matrix and vector

```
>>> m = np.array([[2,3],[1,-1]])
>>> v = np.array([1,2])
>>> m@v
array([ 8, -1])
```

- Notice that the vectors are in row form
- $\begin{pmatrix} 2 & 3 \\ 1 & -1 \end{pmatrix} \cdot (1, \quad 2) = (8, \quad -1)$
- Follows usage of matlab and Mathematica

Operations between Vectors and Matrices

- Transpose with np.transpose or the .T operator

```
>>> m
array([[ 2,  3],
       [ 1, -1]])
>>> m.T
array([[ 2,  1],
       [ 3, -1]])
```

Operations between Vectors and Matrices

- Thus, could have used

```
>>> m@ v.T
array([ 8, -1])
```

Operations between Vectors and Matrices

- We can use this to make a linear transform of a data set

```
def transform(matrix, dataset):
    return (matrix @ dataset.T).T
```

```
mat = np.array([[.1, .2, .3, .4],
                [.2, .2, .3, .4],
                [.1, -.1, .2, 3],
                [3, 2, 1, -2]
               ])
```

Operations between Vectors and Matrices

- Dot-product of two vectors:
  - ```
    v = np.array([1, 2, 3, 4, 5])
    >>> v@v.T
    55
    >>> np.vdot(v,v)
    55
    ```
    ```

Operations between Vectors and Matrices

- Can use linear algebra package in numpy
  - numpy.linalg
  - $$\begin{pmatrix} 1 & 2 \\ 1 & -1 \end{pmatrix}^{10} = \begin{pmatrix} 243 & 0 \\ 0 & 243 \end{pmatrix}$$

```
np.linalg.matrix_power(np.array([[1,2],[1,-1]]),10)
array([[243,   0],
       [  0, 243]])
```

Operations between Vectors and Matrices

- Can calculate matrix inverses
    - Throws LinAlgError if singular

```
>>> np.linalg.inv( np.array([1,-2],[-2,4]) )
Traceback (most recent call last):
…
numpy.linalg.LinAlgError: Singular matrix
```

Operations between Vectors and Matrices

- Can directly solve linear equations
  - Solving $x + 2y = 2, x - y = 3$
    - With solution $x = 8/9, y = -1/3$
  - Gives an error if matrix is not square or singular

```
>>> np.linalg.solve( np.array([[1,2],[1,-1]]) ,
      np.array([2,3]) )
array([ 2.66666667, -0.33333333])
```

NumPy: Universal Array Functions

- There is a plethora of functions that can be applied to a numpy array.
- These are much faster than the corresponding Python functions
- You can find a list in the numpy u-function manual
  - https://docs.scipy.org/doc/numpy/reference/ufuncs.html

NumPy: Universal Array Functions

- There are universal functions around which the operations are wrapped
  - np.add, np.subtract, np.negative, np.multiply, np.divide, np.floor_divide, np.power, np.mod
- The absolute function is
  - abs
  - np.absolute

NumPy:
Universal Array Functions

- Trigonometric functions
  - np.sin, np.cos, np.tan, np.arcsin, np.arccos, np.arctan
- Exponents and logarithms
  - np.log, np.log2 (base 2), np.log10 (base 10)
  - np.expm1 (more exact for small arguments)
  - np.log1p (more exact for small arguments)

NumPy: Universal Array Functions

- Special u-functions:
  - In addition, the submodule scipy.special contains many more specialized functions

NumPy: Universal Array Functions

- Avoid creating temporary arrays
  - If they are large, too much time spent on moving data
  - Specify the array using the 'out' parameter

```
>>> y = np.empty(10)
>>> x = np.arange(1,11)
>>> np.exp(x, out = y)
array([2.71828183e+00, 7.38905610e+00, 2.00855369e+01, 5.45981500e+01,
       1.48413159e+02, 4.03428793e+02, 1.09663316e+03, 2.98095799e+03,
       8.10308393e+03, 2.20264658e+04])
>>> y
array([2.71828183e+00, 7.38905610e+00, 2.00855369e+01, 5.45981500e+01,
       1.48413159e+02, 4.03428793e+02, 1.09663316e+03, 2.98095799e+03,
       8.10308393e+03, 2.20264658e+04])
```

NumPy: Universal Array Functions

- Can use np.min, np.max, sum
- Use np.argmin, np.argmax to find the index of the maximum / minimum element
- Can use np.mean, np.std, np.var, np.median, mp.percentile to get statistics
    - Not the only way, see the scipy module

NumPy: Broadcasting

- Operations can be also made between arrays of different sizes
  - Example 1: adding a scalar (zero-dimensional) to a vector

```
>>> x = np.full(5,1)
>>> x+1
array([2, 2, 2, 2, 2])
```

NumPy: Broadcasting

- Adding a vector to a matrix:
  - Create a matrix

```
>>> matrix = np.arange(1,11).reshape((2,5))
>>> matrix
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

  - Create a vector

```
>>> x = np.arange(1,6)
>>> x
array([1, 2, 3, 4, 5])
```

  - Add them together: The vector has been broadcast to a 2 by 5 matrix by doubling the single row

```
>>> matrix+x
array([[ 2,  4,  6,  8, 10],
       [ 7,  9, 11, 13, 15]])
```

NumPy: Broadcasting

- The broadcast rules: Expand a single coordinate in a dimension in one operand to the value in the other

np.arrange(3) + 5

$$\begin{bmatrix}0 & 1 & 2\end{bmatrix} + \begin{bmatrix}5 & 5 & 5\end{bmatrix} = \begin{bmatrix}5 & 6 & 7\end{bmatrix}$$

np.arrange(9).reshape((3,3)) + np.arrange(3)

$$\begin{bmatrix}0 & 1 & 2\\ 3 & 4 & 5\\ 6 & 7 & 8\end{bmatrix} + \begin{bmatrix}0 & 1 & 2\\ 0 & 1 & 2\\ 0 & 1 & 2\end{bmatrix} = \begin{bmatrix}0 & 2 & 4\\ 3 & 5 & 6\\ 0 & 8 & 10\end{bmatrix}$$

np.arrange(3).reshape((3,1)) + np.arrange(3)

$$\begin{bmatrix}0 & 0 & 0\\ 1 & 1 & 1\\ 2 & 2 & 2\end{bmatrix} + \begin{bmatrix}0 & 1 & 2\\ 0 & 1 & 2\\ 0 & 1 & 2\end{bmatrix} = \begin{bmatrix}0 & 1 & 2\\ 1 & 2 & 3\\ 2 & 3 & 4\end{bmatrix}$$

NumPy: Broadcasting

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading site
- Rule 2: If the shape of two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape
- Rule 3: If in any dimensions the sizes disagree and neither is equal to 1, an error is raised

Neat Example

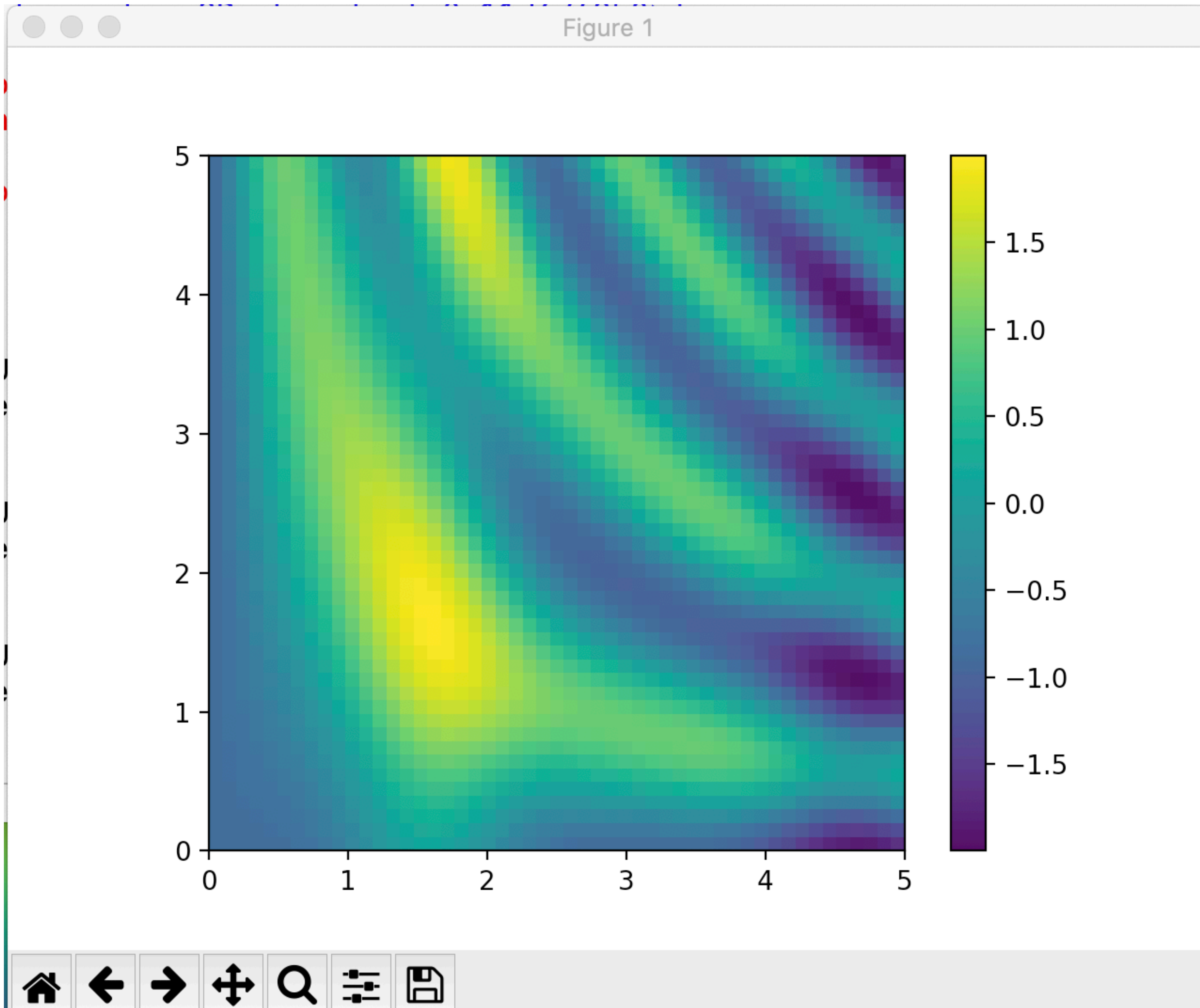
- We combine broadcasting with mathplotlib
    - Using IDLE, we need to call the show function at the end.

NumPy: Broadcasting

- Create a row and a column vector x and y
- Then use broadcasting to combine them for something two-dimensional
- This will get displayed

```
import matplotlib.pyplot as plt
def prob7():
    x = np.linspace(0,5,51)
    y = np.linspace(0,5,51).reshape(51,1)
    z = np.sin(x)**5+np.cos(10+x*y)
    plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],
           cmap='viridis')
    plt.colorbar()
    plt.show()
```

NumPy: Broadcasting



NumPy: Fancy Indexing

- Fancy indexing:
  - Use an array of indices in order to access a number of array elements at once

NumPy: Fancy Indexing

- Example:
  - Create matrix

```
>>> mat = np.random.randint(0,10,(3,5))
>>> mat
array([[3, 2, 3, 3, 0],
       [9, 5, 8, 3, 4],
       [7, 5, 2, 4, 6]])
```

  - Fancy Indexing:

```
>>> mat[(1,2),(2,3)]
array([8, 4])
```

NumPy: Fancy Indexing

- Application:
  - Creating a sample of a number of points
- Create a large random array representing data points

```
>>> mat = np.random.normal(100,20, (200,2))
```

- Select the x and y coordinates by slicing

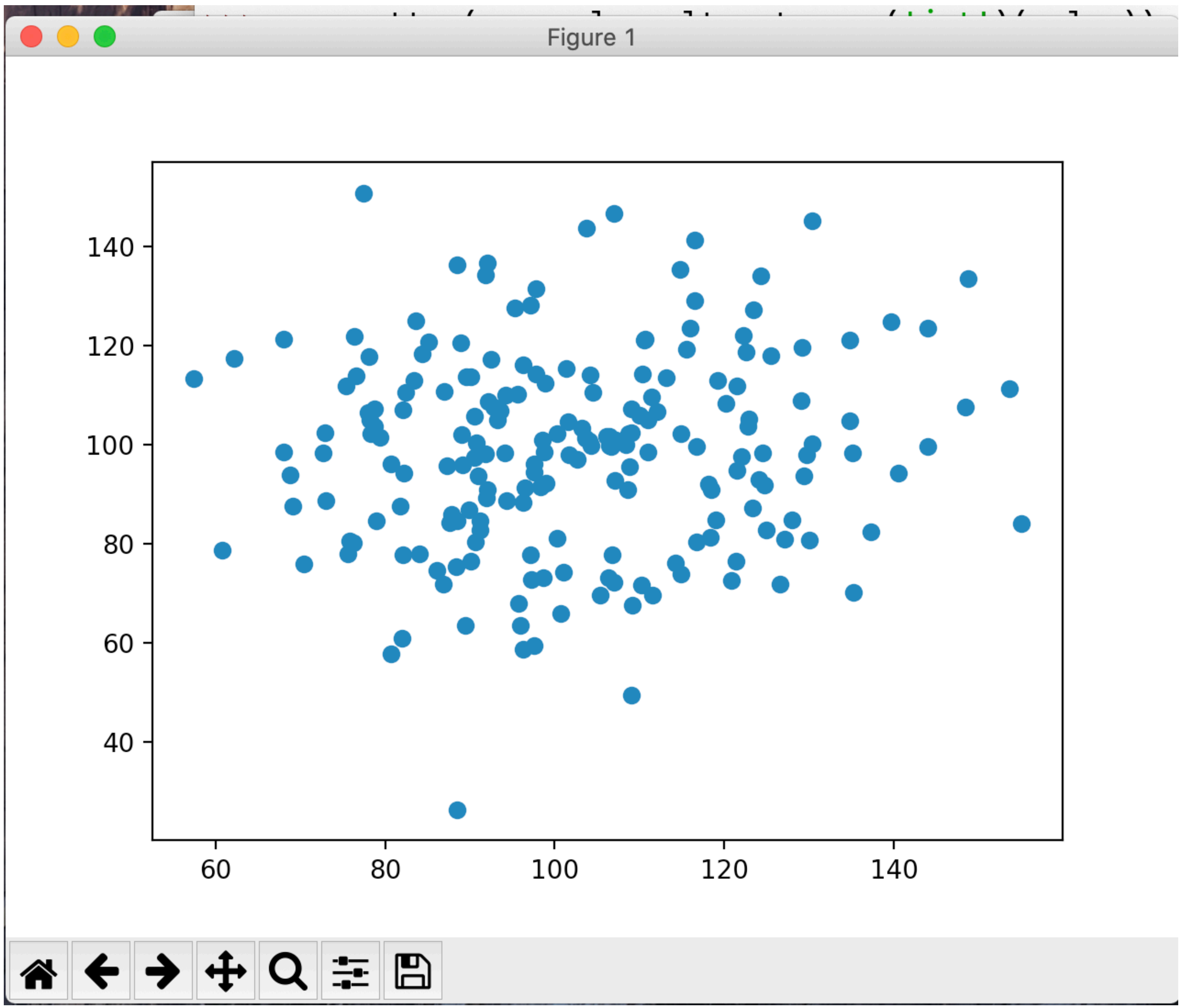
```
>>> x=mat[:,0]
>>> y=mat[:,1]
```

NumPy: Fancy Indexing

- Create a matplotlib figure with a plot inside it

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1,1,1)
>>> ax.scatter(x,y)
>>> plt.show()
```

NumPy: Fancy Indexing



NumPy: Fancy Indexing

- Create a list of potential indices

```
>>> indices = np.random.choice(np.arange(0,200,1),10)
>>> indices
array([ 32,  93, 172, 134,  90,  66, 109, 158, 188,
30])
```

- Use fancy indexing to create the subset of points

```
>>> subset = mat[indices]
```

NumPy: Fancy Indexing