unit_1_numerical_i_clean_FOR_LECTURE

March 25, 2024

0.1 Data Science Fundamentals

1 Lecture 1: Numerical Basics

1.1 ## Numerical arrays and vectorized computation

DSF - University of Glasgow - Chris McCaig - 2023/2024

2 Summary

By the end of this unit you should know:

2.1 1: Why use arrays

2.2 2: Typing and shapes of arrays

2.3 ## 3: Creating, indexing, slicing, joining and rotating

Install Utilities

[ ]:
```
### YOU ONLY NEED TO RUN THIS COMMAND ONCE, TO INSTALL THE LIBRARY: REMOVE '#' AND RUN THE COMMAND

# !pip install -U --no-cache https://github.com/johnhw/jhwutils/zipball/master

### YOU MAY ALSO NEED THESE LIBRARIES

# !pip install -U scikit-image
# !pip install sympy
# !pip install statsmodels
```

[2]:
```
# try:
#     import sympy
#     sympy.init_printing(use_latex='png')
# except:
#     sympy = False

try:
    from Tkinter import *
except ImportError:
```

```
    from tk import *
```

[3]:
```
# various imports we will need
### IF YOU GET AN ERROR WHEN YOU RUN THIS CELL, JUST RUN IT AGAIN WITHOUT␣
 ↪RESTARTING THE KERNEL ###
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import statsmodels.api as sm

from jhwutils.image_audio import (
    play_sound,
    show_image,
    load_image_colour,
    load_image_gray,
    show_image_mpl,
    load_sound
)
from jhwutils.matrices import show_boxed_tensor_latex, print_matrix

%matplotlib inline
plt.rc("figure", figsize=(14.0, 7.0))
```

3 Why use arrays?

3.1 ndarrays

As programmers you have already seen many basic data types (e.g. lists, strings, dictionaries, classes).

The fundamental data type for this course is the **multidimensional numerical array** (of floating point numbers). This is a very powerful data type, although simple in structure, there are great many operations that can be done elegantly with an array.

We will call these arrays **ndarrays** (for *n-dimensional arrays*) or sometimes **tensors** (in reference to the mathematical object which generalises vectors and matrices to higher orders), which some people use.

4 Why use arrays?

4.0.1 Images, sounds, videos

- An *image* is a 2D array of brightness values;
- a *sound* is a 1D array of sound pressure levels;
- Manipulations (e.g. cropping an image, mixing together two sounds) are very straightforward to express in terms of array operations.
- This leads to compact, elegant code that is very efficient.

4.0.2 Scientific data

Scientific data can be conveniently represented as numerical arrays.

Operations we want to do to scientific data are easily expressed as array operations.

4.0.3 3D graphics

3D computer graphics involves manipulating **geometry**.

Geometry is specified as simple geometric shapes, like triangles, made up of points – **vertices** – typically with an $[x, y, z]$ location.

Operations like moving, rotating, scaling of objects are operations on big arrays of these vertices.

Manipulating positions in space efficiently is important in making computer graphics programming work.

Abstraction and elegance By using a numerical array, we can extend operations we apply to single numbers (like integers or floats) to entire arrays of numbers:

e.g. if we have an array of 100 3D positions `pos`, it would be very nice if we could scale all of the points by a factor of 2 (double in size) and move the whole array 100 units right like this:

pos = pos * 2 + [100,0,0]

pos = pos * 2 + [100,0,0]

Code which can express this type of operation without explicit loops is a easier to read and write. Consider the alternative:

```
new_pos = []
for x,y,z in pos:
    new_pos.append((2*x+100, 2*y+0, 2*z+0))
pos = new_pos
```

4.0.4 Mathematical power

Rich set of mathematical abstractions that work on spaces defined over array-valued elements:

- **linear algebra** provides tools to work with 1D arrays (*vectors*) and 2D arrays (*matrices*)
- can be used to solve many difficult problems.
- with types represented as basic types in a programming language working with linear algebraic problems is much easier.

Deep learning Key to deep learning is to represent data as arrays of numbers and to do **all** computations as array operations:

- i.e. perform operations that act on all elements of an array simultaneously.

4.0.5 Vectorisation: one operation, many data

Writing code that acts on arrays of values simultaneously is called **vectorised computation**.

This a special case of **parallel** computing:

- restrict ourselves to numerical operations on fixed size arrays
- modern CPUs have many **vectorised** instructions
- can perform the same operation on many numbers at once

GPUs **Graphics processor units** are by far the most powerful computational units in any modern computer/phone: - essentially supercomputers on a card - can perform calculation much more quickly than CPU

GPUs are array processors **But** effectively big groups of very simple processors - able to deal very well with data in numerical arrays - slow when working with other data structures - anything that can be written as an array operation can be done at lightning speed on a GPU.

- GPUs are basically devices that can do computations on numerical arrays, **and that's it**
- to write (efficient) GPU code, you need to write code in terms of numerical arrays

4.0.6 Efficiency

Numerical arrays are **compact** (they store data in an efficient way) and **computationally efficient** (possible to write code that manipulates arrays quickly)

[Image credit: NOAA, Public domain]

For big, number-focused problems like: * weather simulation * image processing * speech recognition * machine learning

arrays are the best way we have of solving these problems.

In big, text-heavy problems with irregular structure, databases are a more natural structure to store and work with data.

Array types are like entire *spreadsheets in a single variable*, which you can perform standard spreadsheet operations on: * tallying up columns * selecting values which have a certain range * plotting charts * joining together several sheets

Abstraction of array types makes it easy to do complex operations with a standard spreadsheet. And they work on data beyond just 2D tables.

4.1 Typing and shapes of arrays

4.1.1 Vector, matrix, tensor

ndarrays can have different *dimensions*; sometimes called *ranks*, as in a "*rank-3 tensor*" or "3D array" , meaning an array with rows, columns and channels.

[4]:
```
# a vector of length 1
[0]

# a vector of length 3
# (e.g. a position in 3D space)
[1, 2, 3]

# a vector of length 8
```

```
[0, 0, 0, 0, 1, 0, 0, 0]
```

[4]:
```
[0, 0, 0, 0, 1, 0, 0, 0]
```

[5]:
```
show_boxed_tensor_latex(np.array([0, 1, 2, 3]))
```

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

Vector We call a 1D array of values a **vector**: for example:

```
#a 3 element vector
 [1,2,3]
```

Matrix A 2D array of values is called a **matrix**, and is formed of rows and columns:

```
#a 3 x 3 matrix
 [[1,2,3],
  [4,5,6],
  [7,8,9]]
```

Tensors Any array with more than 2 dimensions is just called an **nD array** (**n** **d**imensional array) or sometimes a **tensor**. There isn't a convenient mathematical notation for tensors.

```
# a 2 x 3 x 3 tensor
[[[1,2,3],
  [4,5,6],
  [7,8,9]],

  [10,11,12],
  [13,14,15],
  [16,17,18]]]
```

Often easiest to think of tensors as arrays of matrices or vectors (e.g a 3D tensor is really a stack of 2D matrices, a 4D array is a grid of 2D matrices, a 5D array is a stack of those grids, etc.)

```
# a 2 x 2 x 3 x 3 tensor
# notice how it is really a 2x2 array,
# with each element being a 3x3 subarray
    [[[[ 1  2  3]     [[[19 20 21]
       [ 4  5  6]       [22 23 24]
       [ 7  8  9]]      [25 26 27]]

      [[10 11 12]      [[28 29 30]
       [13 14 15]       [31 32 33]
       [16 17 18]]]     [34 35 36]]]]
```

Typically don't encounter tensors with more than 6 dimensions - these would require enormous amounts of memory to store - don't correspond to many real-world use cases.

4.2 Axes

Often refer to specific dimensions as **axes** or **dimensions**: e.g. - a matrix (a 2D array) has two axes: **rows** (axis 0) and **columns** (axis 1) - a vector has just one axis, axis 0. - a 4D tensor has 4 axes, indexed 0, 1, 2, 3.

Many operations can be selectively applied only on certain axes - very useful way to specify the effect of an operation.

5 Array operations

Unlike data structures you are familiar with (lists, dictionaries, strings), there are **many** operations defined on arrays - some are convenience operations - but there are many fundamental operations as well

5.1 Array operations

- Generalise ideas we have seen on sequence types to sequences across multiple dimensions
- means we can do things like slicing or summing elements, but over *independent dimensions*
- these will be very *very* fast, and memory efficient.

The most important classes of operations that we will cover are:

- **Slice**: slice out rectangular regions, for reading or writing.
  - Chop out second to tenth rows of a 2D matrix: `x[1:9, :]`
  - Set the second column to 0 `x[:,1] = 0`
- **Filter**: find values matching criteria.
  - select elements of x where x is negative `x[x<0]`
- **Reduce**: aggregate across dimensions.
  - compute sum of each column; `np.sum(x, axis=0)`
- **Map**: apply functions or arithmetic operations elementwise
  - add 1 to every element of x `x+1`
  - add x and y `x+y`
  - take the sine of every element of x `np.sin(x)`
- **Concatenate and repeat**:
  - stick x and y together one on top of the other; np.concatenate([x,y], axis=0)
  - repeat x 8 times across the columns; np.tile(x, [1,8])
- **Generate**:
  - create arrays of all zeros: `np.full((8,8),0)`
  - create "counting" arrays: `np.arange(10)`
  - load arrays from files or save them to disk.
- **Reorder**
  - reverse/flip axes
  - sort axes
  - exchange rows/columns (transpose)

5.1.1 No explicit iteration

We operate on arrays without writing explicit iterations wherever possible. This has two effects:

- The code is **much simpler**. Adding two arrays with `c=a+b` is simpler than:

```
for i in range(n):
    for j in range(m):
        c[i][j] = a[i][j] + b[i][j]
```

- The code is **much faster**. The operations can be run in accelerated routines, or with hardware acceleration, e.g. on the GPU.

5.2 Vectors and matrices

As well as being convenient to implement in silicon, arrays correspond to rich mathematical objects. ### Geometry of vectors 1D arrays can be used to represent vectors, which have a mathematical structure that is essential in: - modeling physical systems - building information retrieval systems - machine learning - 3D rendering

5.2.1 Geometry of vectors

Vectors have length and direction: - they can be added, subtracted or scaled - various products are defined on vectors. Arrays are often used to represent vectors; for example a 2D array might be used to store a sequence of vectors (perhaps positions in space), which could be operated on simultaneously.

We write a vector a bold letter lower case symbol: $\mathbf{x}$ (other notations include an arrow or bar above an upper case symbol).

5.2.2 Signal arrays

1D arrays can also be used to represent **signals** - sequences of measurements over time - signals include images, sounds, and other time series - signals can be scaled, mixed, chopped up and rearranged, filtered, and processed in many other ways.

The use of arrays to represent sequences and vectors is not exclusive; some operations use both representations simultaneously. We typically write a signal as $x[t]$.

5.2.3 Algebra of matrices

2D arrays are matrices, which have an algebra: **linear algebra** - a matrix represents a **linear map**, a particular kind of function which operates on vectors (in **vector space**) - the operation of the function is completely defined by the elements of that matrix.

Linear algebra is extremely rich can is used to perform many essential computations - 3D rendering involves matrix operations (e.g. coordinate transformation) - it has many interesting features – multiplication **applies** the map when multiplying with vectors – **composes** the map when multiplying with other matrices.

We write a matrix as an upper case symbol A.

5.2.4 Mathematical operations

We have specialised mathematical operations we can apply to arrays * **Vector operations**: apply geometric effects to vectors: dot product, cross product, norm * e.g. getting the Euclidean length of a vector * **Matrix operations**: linear algebra operations like multiplication, transpose, inverse, matrix exponentials, decompositions * e.g. multiplying together two 3D transformation matrices * **Signal processing operations**: convolution, Fourier transform, numerical gradients, cumulative summation * e.g. blurring an image using a convolution

---

5.3 Statically typed, rectangular arrays: ndarrays

ndarrays (n-dimensional arrays) represent sequences. However, arrays are not like lists.

They have * **fixed, predefined** size (or "shape") * **fixed, predefined** type (all elements have the same type) * they can only hold numbers (typically integers or floating-point) * they are inherently multidimensional * they are required to be "rectangular" – a 2D array must have the same number of columns in each row, for example.

Valid array:

```
1,0,0   # each row has three elements
0,1,0
0,0,1
```

Invalid array:

```
1,0,0,0,0   # 5 elements
0,1         # 2 elements
0,0,1       # 3 elements
```

- type of array has to be specified very precisely; e.g. specify the precision of floating point numbers if we use floats (usually 32 or 64 bits).
- arrays cannot be extended or resized after they have been created
- arrays **are** typically mutable, the values they hold can be changed after creation

5.3.1 Why do we have to type arrays?

Python is dynamically typed and doesn't require types to be specified.

But the elements **within** the array are **not** Python values - stored as a block of "raw numbers" - with information that tells us e.g. the dimensions of the array - the type of the elements - entire array is one *single* value - through some clever syntax it *looks* as if it is made up of individual (Python values) - access and change the values *as if* they were Python values, but internally they are quite different.

Reasons for typing ndarrays: wrapper around raw blocks of memory - this makes them compact and efficient.

- numerical arrays are **much** more efficiently packed into memory
- operations on them can be performed **extremely** quickly

- same is true for other platforms and languages
  - numerical arrays are implemented to be the fastest and smallest possible structure for representing blocks of numbers

6 Practical array manipulation

6.1 NumPy

We use Python with some additional modules - **NumPy** which provides fast array operations - **SciPy** which provides a range of scientific functionality (e.g. statistics) - **Matplotlib** which provides plotting and visualisation functionality.

It has very high performance routines for computations (based on optimized Fortran+C code) and an easy to use API.

Other languages Other languages have numerical libraries which support similar functionality to NumPy - either as built-in features (e.g. Matlab, R, APL, Julia) - or as external libraries (e.g. Eigen for C++, MKL for C, ND4J for Java)

We use NumPy throughout this course (mainly because it is by far the easiest to use option!) - the fundamental ideas generalise outside of the Python ecosystem. - operations we use are standard operations on arrays, not special to NumPy

6.2 Shape and dtype

Every array is characterised by two things: * the type of its elements: the **dtype** (e.g.`float64`) * its **shape**: that is, its dimensions. For example, 32x8

6.2.1 Order

We always discuss the shape of arrays in the order * rows * columns * depth/frames/channels/planes/…

This ordering is important: remember it!

So a 2x3 array means 2 rows, 3 columns:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

A 3x2 array means 3 rows, 2 columns:

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

A 32x8 array has 32 rows of 8 columns each. It has 256 elements (`32*8=256`) in total.

6.2.2 Indexing and axes

We index from **0**, so element [0,1] of an array means *first row, second column*.

The **dimensions** of an array are often called its **axes**. Do not confuse this with axes or dimension of a vector space! The number of dimensions of an array is sometimes called its **rank** (this is distinct from the concept of *rank* in linear algebra, if you are familiar with this).

- A scalar is a rank 0 tensor
- A vector is a rank 1 tensor (1 dimensional array)
- A matrix is a rank 2 tensor (2 dimensional array)
- A stack of matrices is a rank 3 tensor (3 dimensional array)
- and so on…

6.3 Dtypes

dtype just stands for the *d*ata *type*, and it is the data type of every element in an array (what kind of number it is). For the moment, we will assume we get floating point elements in our arrays: we'll discuss this in more detail later. Every element has the same *dtype*; it applies to the whole array.

Common *dtypes* are: * `float64` double-precision float numbers * `float32` single-precision float numbers * `int32` signed 32 bit integers * `uint8` unsigned 8 bit integers

though there are many more.

We can interrogate an array to see its *shape* and *dtype*:

[6]:
```
x = np.zeros((3,3))
print('dtype =',x.dtype)
print('shape = ', x.shape)
```

```
dtype = float64
shape =  (3, 3)
```

6.4 Images and sounds

Images and sounds are classic examples of arrays. They are naturally manipulated using array abstractions. For example, we can load an image from disk, and its `shape` will be its resolution:

[7]:
```
from matplotlib.image import imread

img = imread('imgs/rocks.png')
print(type(img),",",img.dtype,",",'shape = ',img.shape)
```

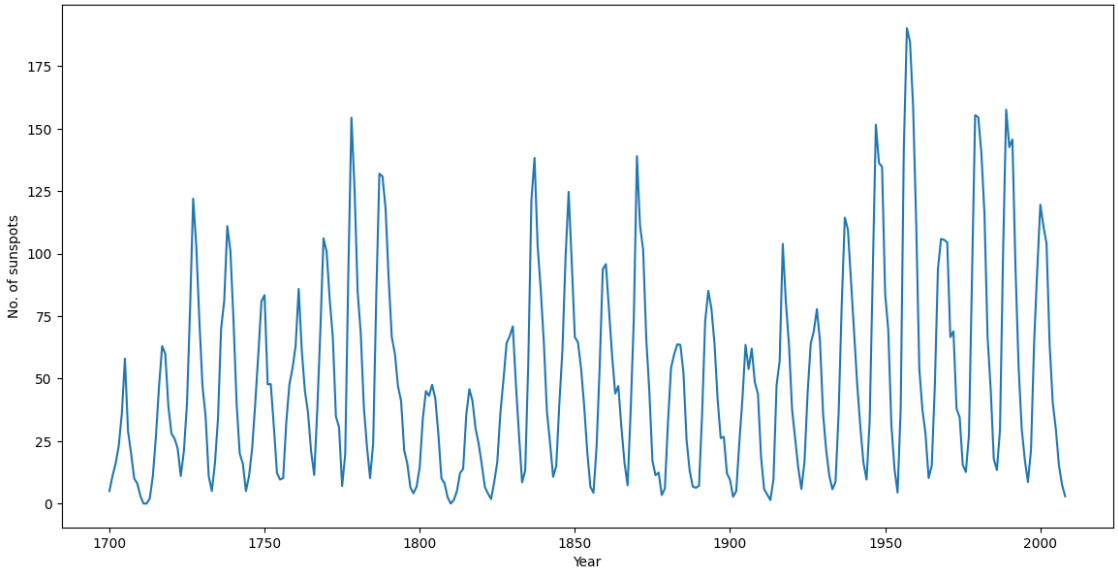
```
<class 'numpy.ndarray'> , float32 , shape =  (287, 400, 3)
```

As we will see in detail later, we can for example plot this data array in one go:

[8]:
```
sunspots = np.array(sm.datasets.sunspots.load().data)

import matplotlib.pyplot as plt
```

```
%matplotlib inline

fig = plt.figure()
ax = fig.add_subplot(1,1,1)

ax.plot(sunspots[:,0], sunspots[:,1])
ax.set_xlabel("Year")
ax.set_ylabel("No. of sunspots")
```

[8]: Text(0, 0.5, 'No. of sunspots')



7 Array ops: creation, indexing, slicing, joining and rotating

7.1 Creating arrays

7.1.1 Converting and copying: np.array

New arrays can be created in several ways:

- Converted from another sequence type, like a list: `np.array()` does this.
- Created blank and filled with some value. This is often essential in creating temporary variables, for example to accumulate results into.
- Filled with random values.
- Loaded from disk.

`np.array()` takes a sequence and converts it into an array; this can be, for example, a list. It works for multidimensional arrays as well, given nested sequences.

7.2 Mutability and copying

`np.array()` can take any sequence, including another ndarray. So it can be used to copy arrays:

This is important, because NumPy arrays are **mutable**, and if several variables refer to the **same** array, the effects might not be what you expect:

[9]:
```
x = np.array([1,2,3])
z = x
x[0] = 0
print(x)
print(z) # huh?
```

```
[0 2 3]
[0 2 3]
```

We need to explicitly copy arrays if we want to work on a new array:

[10]:
```
x = np.array([1,2,3])
y = np.array(x) # copy
z = x.copy() # same thing as using np.array(x)
x[0] = 0
print(x) # 0 2 3
print(y) # still 1 2 3
print(z)
```

```
[0 2 3]
[1 2 3]
[1 2 3]
```

7.2.1 Ragged arrays (bad!)

What happens if we **don't** follow the rectangular array rule?

[11]:
```
# This **looks** like it should work, but it doesn't do what we want
bad_array = np.array([
                [1.0, 2.0, 3.0, 6.0, 7.0],
                [0.0, 0.5, 0.0],  # ragged!
                [5.0, 0.0, 10.0]
               ])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[11], line 2
      1 # This **looks** like it should work, but it doesn't do what we want
----> 2 bad_array = np.array([
      3                 [1.0, 2.0, 3.0, 6.0, 7.0],
      4                 [0.0, 0.5, 0.0],  # ragged!
      5                 [5.0, 0.0, 10.0]
      6                ])
```

```
ValueError: setting an array element with a sequence. The requested array has a␣
↪inhomogeneous shape after 1 dimensions. The detected shape was (3,) +␣
↪inhomogeneous part.
```

In general, *never* create ragged arrays. Always make sure you are creating rectangular arrays!

8 Blank arrays

There are a number of methods all of which do essentially the same thing; allocate a new array with a given shape, and possibly fill it with a value.

- `np.empty(shape)`, which allocates memory for an array, but does not initialise it
- `np.zeros(shape)`, which initialises all elements to 0
- `np.ones(shape)`, which initialises all elements to 1
- `np.full(shape, value)` which initialises all elements to `value`

These are just calling `np.empty(shape)` to create a new array and then filling it with a given value.

For example, `x = np.ones((5,8))` is exactly the same as:

[12]:
```
x = np.empty((5,8))
x[:,:] = 1
x
```

[12]:
```
array([[1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.]])
```

8.0.1 Blank like

Similarly, we can create blank arrays with the same shape and dtype as an existing array using the `_like` variants. `y = np.zeros_like(x)` is exactly the same as:

```
y = np.empty(x.shape, dtype=x.dtype)
y[:] = 0.0
```

[13]:
```
y = np.zeros_like(x)
print(y)
```

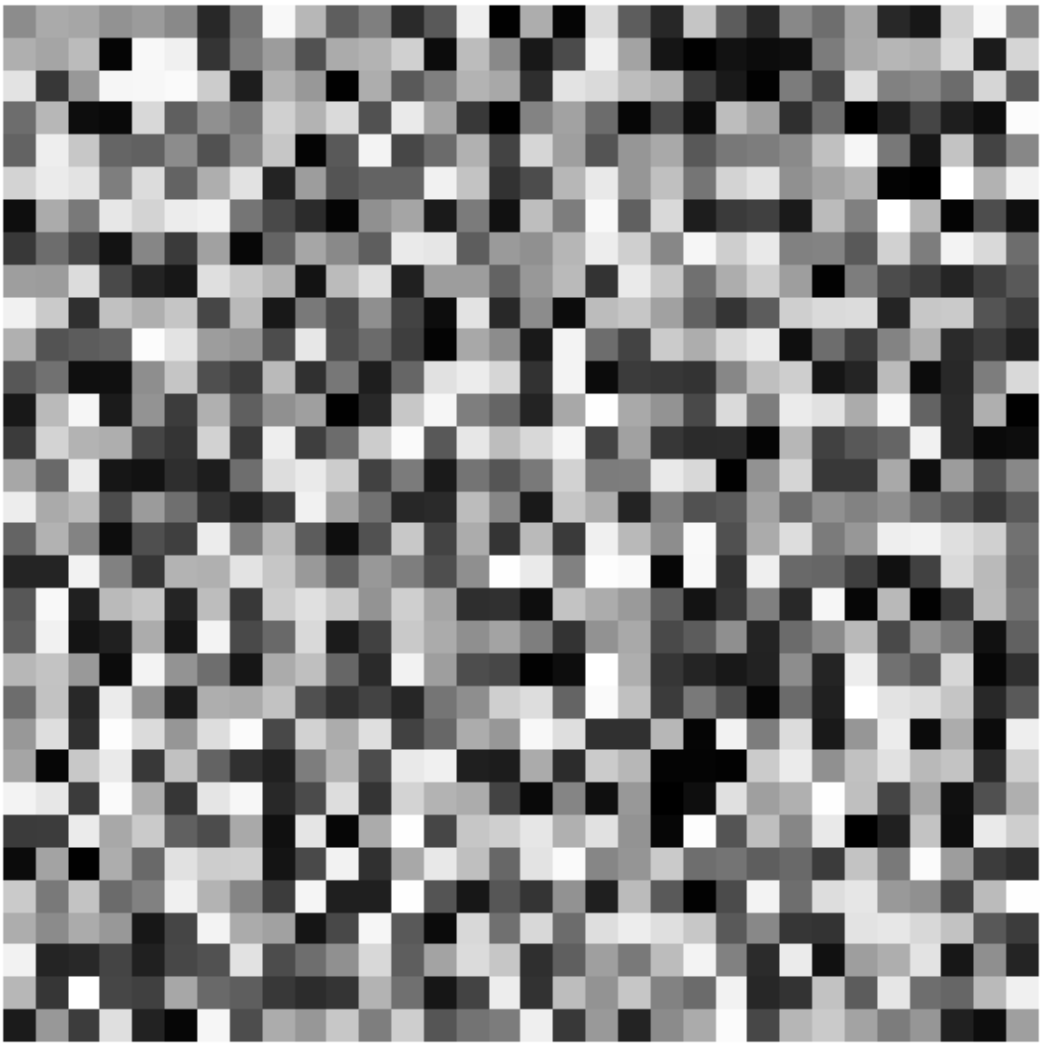
```
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

8.0.2 Random arrays

We can also generate random numbers to fill arrays. Many algorithms use arrays of random numbers as their basic "fuel".

- `np.random.randint(a,b,shape)` creates an array with uniform random *integers* between a and (excluding) b
- `np.random.uniform(a,b,shape)` creates an array with uniform random *floating point* numbers between a and b
- `np.random.normal(mean,std,shape)` creates an array with normally distributed random floating point numbers between with the given mean and standard deviation.

[14]:
```
# a 32x32 block of random numbers
show_image_mpl(np.random.uniform(0, 1, (32, 32)))
```



9 Ranges

9.1 arange

We can create a vector of increasing values using `arange` (**a**rray **range**), which works like the built in Python function `range` does, but returns an 1D array (a vector) instead of a list.

`np.arange()` takes one to three parameters: * `np.arange(end)` – returns a vector of numbers 0..end-1 * `np.arange(start, end)` – returns a vector of numbers start..end-1 * `np.arange(start, end, step)` –returns a vector of numbers start..end-1, incrementing by step (which may be **negative** and/or **fractional**!)

[15]:
```
np.arange(10)
```

[15]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[16]:
```
np.arange(5,10)
```

[16]:
```
array([5, 6, 7, 8, 9])
```

[17]:
```
np.arange(0, 10, 2)
```

[17]:
```
array([0, 2, 4, 6, 8])
```

9.1.1 Linspace

`np.arange` is useful for generating evenly spaced values, but it is parameterised in a form that can be awkward.

`np.linspace(start, stop, steps)` is a much easier to use alternative. `linspace` stands for "**lin**early **space**d", and it generates `steps` values between `start` and `stop` **inclusive**.

[18]:
```
np.linspace(0,10,11) # careful -- end is inclusive, so we need 11 steps if we want integer spacing
```

[18]:
```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

[19]:
```
# show a gradient, using linspace to generate an array, and then duplicate it onto 20 rows
show_image(np.tile(np.linspace(0,1,100), (20,1)))
```

9.1.2 Loading and saving arrays

I/O with arrays is a critical part of any numerical computation system. There are a huge number of ways to store and recall arrays, including: * simple text formats like CSV (comma separated

values) * binary formats for storing single or multiple arrays like mat and npz * specialised scientific data formats like HDF5 (often used for huge datasets) * domain-specific formats like images (png, jpg, etc.), sounds (wav, mp3, etc.), 3D geometry (obj, …)

Text files

- `np.loadtxt(fname)` and
- `np.savetxt(arr, fname)`

work on simple text files.

[20]:
```
# this just turns off scientific notation, so the output looks sensible!
np.set_printoptions(suppress=True)

# load and print, in a single array
sunspots = np.loadtxt("data/sunspots.csv", delimiter=',')
print(sunspots)

np.savetxt("data/sunspots_out.csv", sunspots )
```

```
[[   1.         1749.           58.        ]
 [   2.         1749.08333333   62.6       ]
 [   3.         1749.16666667   70.        ]
 …
 [2818.         1983.75         55.8       ]
 [2819.         1983.83333334   33.3       ]
 [2820.         1983.91666667   33.4       ]]
```

10 Slicing and indexing arrays

Arrays can be indexed like lists or sequences (in Python, this uses square brackets []), but arrays can have **multidimensional** indices. These are indices which are really tuples of values.

This means we write the variable, with the index in square brackets, where the index might have comma separated values. Indices start at **zero**!

[21]:
```
x = np.arange(10)
x[0] # simple indexing; one dimension
```

[21]:
```
0
```

[22]:
```
sunspots[0, 0] # first row, first column
```

[22]:
```
1.0
```

[23]:
```
sunspots[-1, -1] # last row, last column
```

[23]:
```
33.4
```

[24]:
```
sunspots[5, 2]
```

[24]:
```
83.5
```

Indexing, and its counterpart slicing, are two of the most important array operations.

The general format follows the same principles as `arange()`, taking 0-3 parameters separated by a `:`

```
start : stop : step
```

Where `start` is the index to start from, `stop` is the end, and `step` is the jump to make between each step. **Any of these parts can be omitted**.

- If there is no colon, this specifies a specific index, for example x[0] or x[18]
- If there is one colon, this is a range; for example x[2:5] or x[:4] or x[0:]
- If there are two colons, this is a range with a step, like x[0:10:2], or x[:10:2] or x[::-1]
- If `start` is missing, it defaults to 0
- If `end` is missing, it defaults to the last element
- If `step` is missing, it defaults to 1. You don't need to include the second colon if you are omitting step, though it's not an error to do so.

Negative indices Negative indices mean *counting from the end*; so `x[-1]` is the last element, `x[-2]` is the second last, etc.

If we specify an axis as an index (no range), we get back a **slice** of that array, with fewer dimensions. If `x` is 2D, then `x[0,:]` is the first row (a 1D vector), and `x[:,0]` is the first column (a 1D vector).

[25]:
```
print(sunspots[:, 1])  # second column (remember 0-indexing!)
```

```
[1749.         1749.08333333 1749.16666667 … 1983.75       1983.83333334
 1983.91666667]
```

[26]:
```
print(sunspots[0, :])  # first row (note the row=0, column=any)
```

```
[   1. 1749.   58.]
```

[27]:
```
# first ten rows, second column onwards (start specified, but not stop)
print(sunspots[0:10, 1:])
```

```
[[1749.         58.        ]
 [1749.08333333 62.6       ]
 [1749.16666667 70.        ]
 [1749.25       55.7       ]
 [1749.33333333 85.        ]
 [1749.41666667 83.5       ]
 [1749.5        94.8       ]
 [1749.58333333 66.3       ]
```

```
 [1749.66666667   75.9       ]
 [1749.75         75.5       ]]
```

[28]:
```
# every tenth row, up to row 100, second column onwards
# note this uses the full start:stop:step slicing
print(sunspots[0:100:10, 1:])
```

```
[[1749.           58.        ]
 [1749.83333333  158.6       ]
 [1750.66666667   91.2       ]
 [1751.5          66.3       ]
 [1752.33333333   59.7       ]
 [1753.16666667   45.7       ]
 [1754.            0.        ]
 [1754.83333333   13.2       ]
 [1755.66666667   17.8       ]
 [1756.5           3.6       ]]
```

[29]:
```
# whole array, every sixtieth row. Note that start and stop are *both* omitted␣
 ↪here
# and only step is given.
print(sunspots[::60, :])
```

```
[[   1.  1749.    58. ]
 [  61.  1754.     0. ]
 [ 121.  1759.    48.3]
 [ 181.  1764.    59.7]
 [ 241.  1769.    73.9]
 [ 301.  1774.    46.8]
 [ 361.  1779.   114.7]
 [ 421.  1784.    13. ]
 [ 481.  1789.   114. ]
 [ 541.  1794.    45. ]
 [ 601.  1799.     1.6]
 [ 661.  1804.    45.3]
 [ 721.  1809.     7.2]
 [ 781.  1814.    22.2]
 [ 841.  1819.    32.5]
 [ 901.  1824.    21.6]
 [ 961.  1829.    43. ]
 [1021.  1834.     4.9]
 [1081.  1839.   107.6]
 [1141.  1844.     9.4]
 [1201.  1849.   156.7]
 [1261.  1854.    15.4]
 [1321.  1859.    83.7]
 [1381.  1864.    57.7]
 [1441.  1869.    60.9]
```

```
 [1501.  1874.    60.8]
 [1561.  1879.     0.8]
 [1621.  1884.    91.5]
 [1681.  1889.     0.8]
 [1741.  1894.    83.2]
 [1801.  1899.    19.5]
 [1861.  1904.    31.6]
 [1921.  1909.    56.7]
 [1981.  1914.     2.8]
 [2041.  1919.    48.1]
 [2101.  1924.     0.5]
 [2161.  1929.    68.9]
 [2221.  1934.     3.4]
 [2281.  1939.    80.3]
 [2341.  1944.     3.7]
 [2401.  1949.   119.1]
 [2461.  1954.     0.2]
 [2521.  1959.   217.4]
 [2581.  1964.    15.3]
 [2641.  1969.   104.4]
 [2701.  1974.    27.6]
 [2761.  1979.   166.6]]
```

[30]:
```
print(sunspots[0, 1])  # first entry in second column
# note that there is no colon here; this indexes rather than slices
```

```
1749.0
```

10.0.1 Slicing versus indexing

- **Slicing** does not change the rank of an array. It selects a rectangular subset with the same number of dimensions.
- **Indexing** reduces the rank of an array (usually). It selects a rectangular subset where one dimension is a singleton, and removes that dimension.

[31]:
```
print(sunspots[0])  # indexes first row, does reduce rank
print(sunspots[0].shape)  # indexes first row, does reduce rank
```

```
[   1. 1749.   58.]
(3,)
```

[32]:
```
print(sunspots[0:1])  # slices first row, does not reduce rank
print(sunspots[0:1].shape)  # slices first row, does not reduce rank
```

```
[[   1. 1749.   58.]]
(1, 3)
```

10.0.2 Sounds

Likewise, we can chop out a bit of a sound by slicing, or even reverse it and pitch it up:

[33]:
```
snd = load_sound("sounds/guitar.wav")
play_sound(snd)  # original
```

```
/home/chris/.local/lib/python3.10/site-packages/jhwutils/image_audio.py:73:
WavFileWarning: Chunk (non-data) not understood, skipping it.
  sr, sound = scipy.io.wavfile.read(wav_file)
```

```
<IPython.lib.display.Audio object>
```

[34]:
```
play_sound(snd[4000:18000]) # slice
```

```
<IPython.lib.display.Audio object>
```

[35]:
```
play_sound(snd[::-1]) # backwards
```

```
<IPython.lib.display.Audio object>
```

[36]:
```
play_sound(snd[::2]) # double speed (horrible interpolation+aliasing artifacts)
```

```
<IPython.lib.display.Audio object>
```

10.1 Boolean tests

We can do any test (like equality, greater than, etc.) on arrays as well: the result is a **Boolean array**, with the same shape as the original array (despite how they appear, these are actually numeric arrays internally):

[37]:
```
x2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

[38]:
```
print_matrix("x_2", x2)

print_matrix("x_2>5", np.where(x2>5, 1, 0))
```

```
x_2
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
x_2>5
 [[0 0 0]
 [0 0 1]
 [1 1 1]]
```

[39]:
```
y2 = np.array([[1, -1, 1], [-1, 1, -1], [1, -1, 1]])
print_matrix("x_2", x2)
print_matrix("y_2", y2)
print_matrix("x_2>y_2", np.where(x2 > y2, 1, 0))
```

```
x_2
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
y_2
 [[ 1 -1  1]
 [-1  1 -1]
 [ 1 -1  1]]
x_2>y_2
 [[0 1 1]
 [1 1 1]
 [1 1 1]]
```

[40]:
```
# we can freely combine comparisons and arithmetic
print_matrix("x_2 == y_2+2", np.where(x2 == y2+2, 1, 0))
```

```
x_2 == y_2+2
 [[0 0 1]
 [0 0 0]
 [0 0 0]]
```

10.2 Rearranging arrays

Arrays can be transformed and reshaped; this means that they keep the same elements, but the arrangements of the elements are changed. For example, the sequence

```
[1,2,3,4,5,6]
```

could be rearranged into

```
[6,5,4,3,2,1]
```

which has the same elements but now ordered backwards.

These operations are often very useful to rearrange arrays so that broadcasting operations can be carried out effectively.

[41]:
```
x = np.arange(6)+1
print_matrix("x", x)
```

```
x
 [[1 2 3 4 5 6]]
```

[42]:
```
print_matrix("x_{r}", x[::-1]) # note slicing used to reverse
```

```
x_{r}
 [[6 5 4 3 2 1]]
```

10.3 Transposition

A particularly useful transformation of an array is the **transpose** which exchanges rows and columns (this isn't the same as rotating 90 degrees!). There is special syntax for this because it is so often used:

We write `x.T` to get the transpose of `x`.

[43]:
```
# square example
x = np.array([[1, 2, 3], [0, 0, 0], [4, 5, 6]])
print_matrix("x", x)
```

```
x
 [[1 2 3]
 [0 0 0]
 [4 5 6]]
```

[44]:
```
print_matrix("x^T", x.T)
```

```
x^T
 [[1 0 4]
 [2 0 5]
 [3 0 6]]
```

[45]:
```
# non square example
y = np.array([[1,2], [3,4], [5,6]])
print_matrix("y", y)
```

```
y
 [[1 2]
 [3 4]
 [5 6]]
```

[46]:
```
print_matrix("y^T", y.T)
```

```
y^T
 [[1 3 5]
 [2 4 6]]
```

Transposition has *no effect* on a 1D array, and it reverses the order of all dimensions in >2D arrays.

[47]:
```
tensor = np.zeros((10,5,60,2))  # 10 x 5 x 60 x 2 array

print("Original shape\t\t", tensor.shape)
print("Transposed shape\t", tensor.T.shape) # dimensions in reverse order
```

```
Original shape          (10, 5, 60, 2)
Transposed shape        (2, 60, 5, 10)
```

[48]:
```
test = np.zeros(5, dtype=int)
print_matrix("test",test)
print_matrix("test^T",test.T)
```

```
test
 [[0 0 0 0 0]]
test^T
 [[0 0 0 0 0]]
```

Note that transposing is a very fast operation – it does not (normally) copy the array data, but just changes how it is accessed, and thus has virtually no time penalty, and completes in O(1) time. We will discuss how this is possible in the next lecture.

[49]:
```
## show that transpose is fast, and does not depend on array sizze
x300 = np.zeros((300,300))
x3 = np.zeros((3,3))
```

[50]:
```
%%timeit
x3.T
```

```
67.3 ns ± 1.6 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

[51]:
```
%%timeit
x300.T
```

```
66.5 ns ± 1.5 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

10.3.1 Flip, rotate

As well as transposition, arrays can also be flipped and rotated in a single operation using indexing (there are also convenience functions like `fliplr` and `rot90` but we will keep it simple here):

[52]:
```
print_matrix("x", x)

# left-right flip
print_matrix("\\text{fliplr}(x)", x[:, ::-1]) # ::-1 --> reverse order
# up-down flip
print_matrix("\\text{flipud}(x)", x[::-1, :])
```

```
x
 [[1 2 3]
 [0 0 0]
 [4 5 6]]
\text{fliplr}(x)
 [[3 2 1]
 [0 0 0]
 [6 5 4]]
\text{flipud}(x)
 [[4 5 6]
```

```
 [0 0 0]
 [1 2 3]]
```

[53]:
```
# rotate 90 (same as transpose + flipud)
print_matrix("\\text{rot90}(x)", x.T[::-1, :])
```

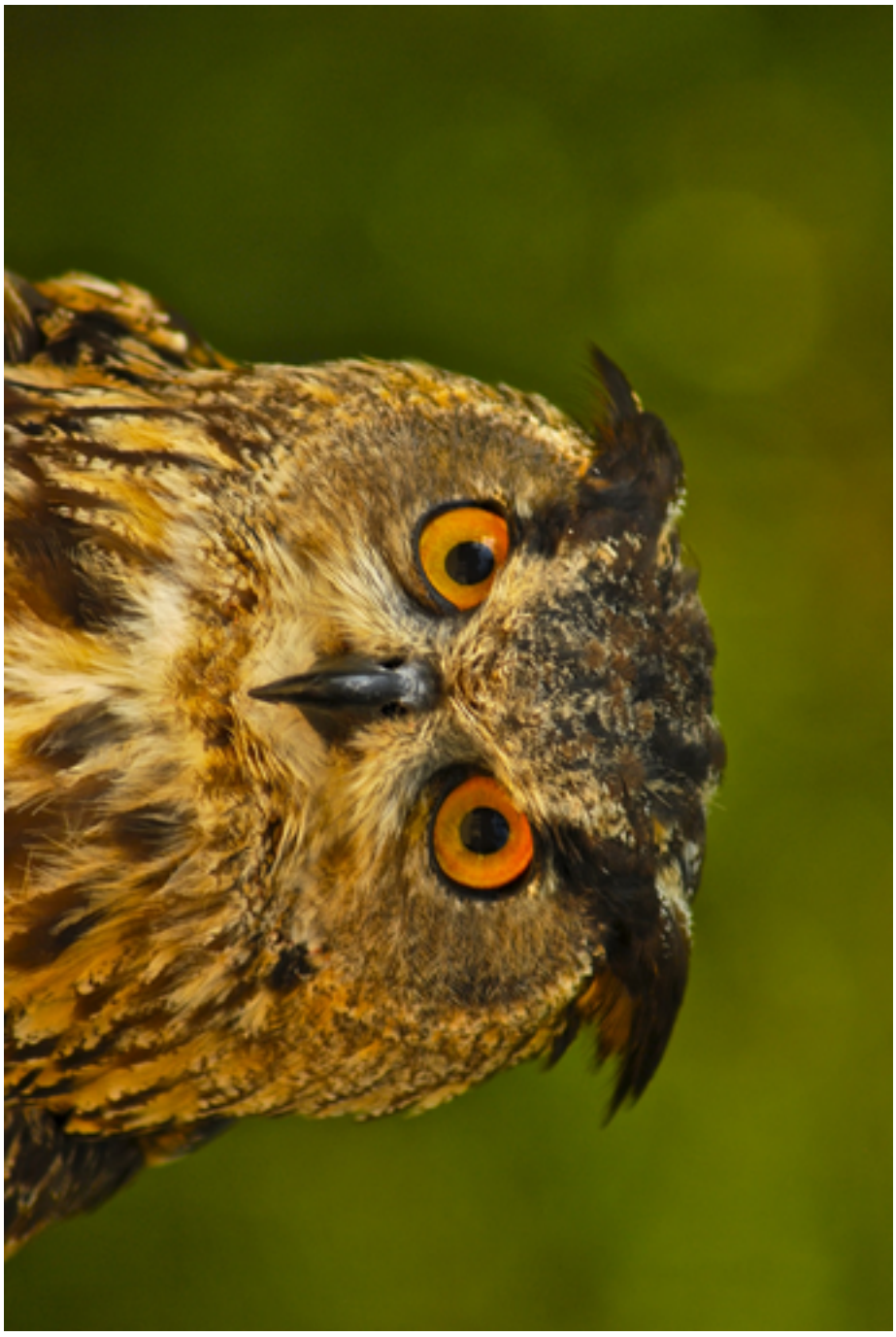
```
\text{rot90}(x)
 [[3 0 6]
 [2 0 5]
 [1 0 4]]
```

[54]:
```
owl = load_image_colour("imgs/owl.png")
show_image(owl, width="50%")
```

[55]:
```
# flip up-down
show_image(owl[::-1, :], width="50%")
```

[56]:
```
# flip left-right
show_image(owl[:,::-1], width="50%")
```

[57]:
```
# rotate 90
show_image(owl[::-1, :].swapaxes(0,1), width="50%")
```



Symmetric owls

[58]:
```
# split the owl into a left and right half
h,w,d = owl.shape

left = owl[:, :w//2, :]  # //2 just means integer division by 2 (i.e. no␣
 ↪floating point part)
right = owl[:, w//2:, :]
## we'll see concatenate in a moment
show_image(np.concatenate([left, left[:,::-1]], axis=1), width="50%")
#show_image(np.concatenate([right[:,::-1], right], axis=1), width="50%")
```

11 Cut+tape operations

11.1 Joining and stacking

We can also join arrays together. But unlike simple structures like lists, we have to explicitly state on which **dimension** we are going to join. And we must adhere to the rule that the output array has rectangular shape; we can't end up with a "ragged" array. (arrays are *always* rectangular)

11.1.1 concatenate and stack

Because arrays can be joined together along different axes, there are two distinct kinds of joining: * We can use `concatenate` to join along an *existing* dimension; * or `stack` to stack up arrays along a *new dimension.*

[59]:
```
x = np.array([1,2,3,4])
y = np.array([5,6,7,8])
print_matrix("x", x)
print_matrix("y", y)
```

```
x
 [[1 2 3 4]]
y
 [[5 6 7 8]]
```

[60]:
```
print_matrix("\\text{stack}(x, y)", np.stack([x,y]))
```

```
\text{stack}(x, y)
 [[1 2 3 4]
 [5 6 7 8]]
```

[61]:
```
print_matrix("\\text{concatenate}(x,y)", np.concatenate([x,y]))
```

```
\text{concatenate}(x,y)
 [[1 2 3 4 5 6 7 8]]
```

[62]:
```
x=np.arange(4)
x.shape=(2,2)
print(np.concatenate((x,x),axis=0),"\n")
print(np.concatenate((x,x),axis=1),"\n","\n")
print(np.stack((x,x)))
```

```
[[0 1]
 [2 3]
 [0 1]
 [2 3]]

[[0 1 0 1]
 [2 3 2 3]]


[[[0 1]
  [2 3]]

 [[0 1]
  [2 3]]]
```

If we `concatenate` when we have multiple dimensions, we can specify explicitly which axis to join on:

[63]:
```
# when we have multiple dimensions, we can specify explicitly
# which axis to join on
x = np.zeros((3, 3))
```

```
y = np.ones((3, 3))
print_matrix("x", x)
print_matrix("y", y)
```

```
x
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
y
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

[64]:
```
print_matrix("\\text{concatenate}_0(x,y)", np.concatenate([x,y], axis=0))
print_matrix("\\text{concatenate}_1(x,y)", np.concatenate([x,y], axis=1))
```

```
\text{concatenate}_0(x,y)
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
\text{concatenate}_1(x,y)
 [[0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1.]]
```

11.1.2 2D matrix stacking shorthand

As a shorthand, there are three defined stacking operations for specific axes when working with 2D matrices: * `np.hstack()` stacks horizontally * `np.vstack()` stacks vertically * `np.dstack()` stacks "depthwise" (i.e. one matrix on top of another)

All of these operate on 2D matrices

[65]:
```
print_matrix('\\text{hstack}',
             np.hstack([x, y]))  # same as np.concatenate([x,y], axis=0)
```

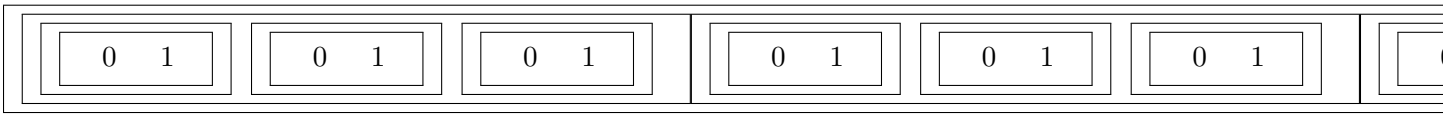
```
\text{hstack}
 [[0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1. 1.]]
```

[66]:
```
print_matrix('\\text{vstack}',
             np.vstack([x, y]))  # same as np.concatenate([x,y], axis=1)
```

```
\text{vstack}
 [[0. 0. 0.]
```

```
 [0. 0. 0.]
 [0. 0. 0.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

[67]:
```
show_boxed_tensor_latex(np.dstack([x, y]))  # same as np.stack([x,y])
```



11.2 Tiling

We often need to be able to **repeat** arrays. This is called **tiling** and `np.tile(a, tiles)` will repeat `a` in the shape given by `tiles`, joining the result together into a single array.

[68]:
```
eye = np.array([[1., 0.], [0., -1.]])
print_matrix("e", eye)
```

```
e
 [[ 1.  0.]
 [ 0. -1.]]
```

[69]:
```
print("Repeated 4 times, columns")
print_matrix("e_{1,4}", np.tile(eye, (1,4)))
```

```
Repeated 4 times, columns
e_{1,4}
 [[ 1.  0.  1.  0.  1.  0.  1.  0.]
 [ 0. -1.  0. -1.  0. -1.  0. -1.]]
```

[70]:
```
print("Repeated 4 times, rows")
print_matrix("e_{4,1}", np.tile(eye, (4,1)))
```

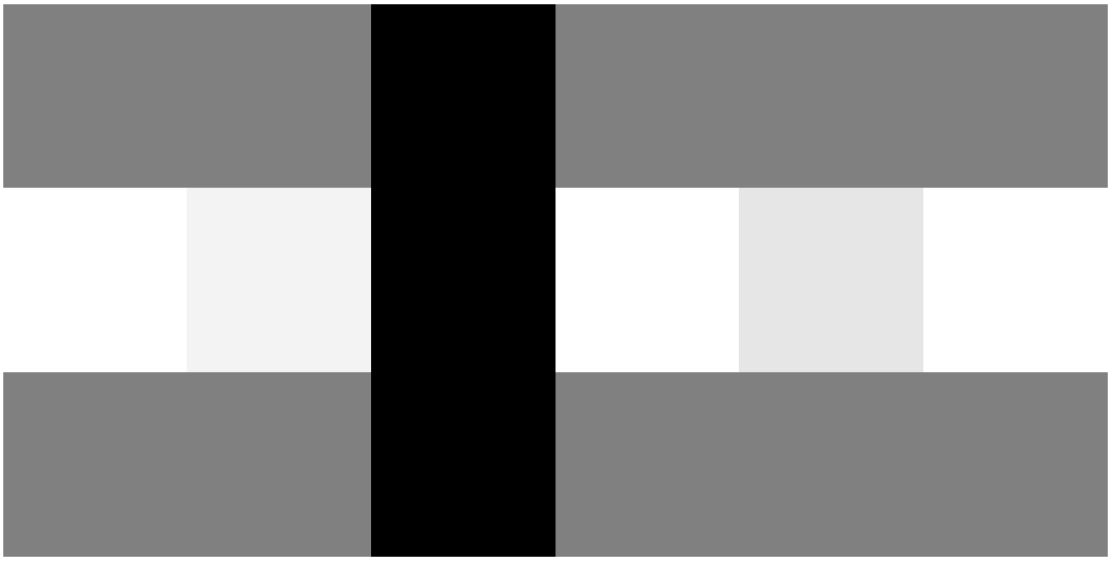
```
Repeated 4 times, rows
e_{4,1}
 [[ 1.  0.]
 [ 0. -1.]
 [ 1.  0.]
 [ 0. -1.]
 [ 1.  0.]
 [ 0. -1.]
 [ 1.  0.]
 [ 0. -1.]]
```

[71]:
```
print("Repeated 2x2x2x2 times")

show_boxed_tensor_latex(np.tile(eye, (2,2,2,2)))
```

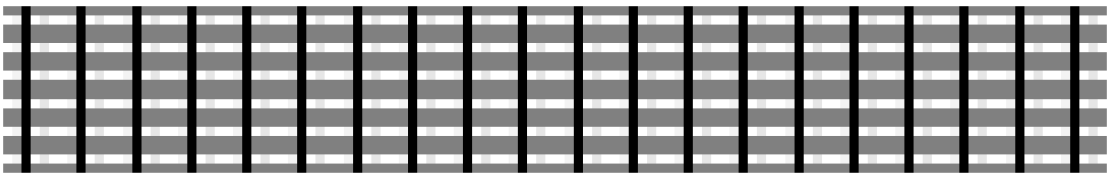
```
Repeated 2x2x2x2 times
```

$$\left[\begin{array}{cccc|cccc|cccc|cccc} 1 & 0 & 1 & 0 & 0 & -1 & 0 & -1 & 1 & 0 & 1 & 0 & 0 & -1 & 0 & -1 \end{array}\right]$$

[72]:
```
fret = np.array([[0.5, 0.5,0,0.5, 0.5,0.5],
                 [1,0.95, 0, 1,0.9,1],
                 [0.5, 0.5, 0, 0.5, 0.5,0.5]])
show_image_mpl(fret)
```



[73]:
```
show_image_mpl(np.tile(fret, (6, 20)))
# <-- insert rest of guitar here
```



[74]:
```
# 3x3 grid of owls
show_image(np.tile(owl, (3,3,1)))
```

11.3 Stretching sounds

We can use tiling to time stretch sounds:

[75]:
```
# choose an even sized segment

guitar = load_sound("sounds/guitar.wav")
play_sound(guitar)
```

```
<IPython.lib.display.Audio object>
```

[76]:
```
snd = guitar[:48000 * 4]
# split into 4800 segment wide blocks
snd_s = snd.reshape(-1, 2)
snd_s = np.tile(snd_s, (1, 2)).ravel()
snd_s = snd_s.reshape(-1, 4800)
snd_s = snd_s[::2,:].ravel()

# tile each block 4 times, then read out the whole thing
# in one long thread
```

```
play_sound(snd_s)
```

```
<IPython.lib.display.Audio object>
```

[77]:
```
# we can actually fix the clicking easily
import scipy.signal
w = 1200
slow = 3
snd_s = snd.reshape(-1, w)
snd2 = np.concatenate([np.zeros(w // 2, ), snd, np.zeros(w // 2, )])
snd_s2 = snd2.reshape(-1, w)

n = scipy.signal.hann(w)  # some magic
a = np.tile(snd_s * n, (1, slow)).ravel()
b = np.tile(snd_s2 * n, (1, slow)).ravel()
play_sound((a + b[w // 2:-(slow * w - w // 2)]))
```

```
<IPython.lib.display.Audio object>
```

12 Selection and masking

Comparisons between arrays result in Boolean arrays, as we have seen:

[78]:
```
x = np.array([1, 2, 3, 4])
y = np.array([2, 2, 2, 2])
print_matrix("x", x)

print_matrix("y", y)
print_matrix("x>y", np.where(x > y, 1, 0))
```

```
x
 [[1 2 3 4]]
y
 [[2 2 2 2]]
x>y
 [[0 0 1 1]]
```

These Boolean arrays have many useful applications in **selecting** specific data, or alternatively **masking** specific data. Selection and masking are basic operations.

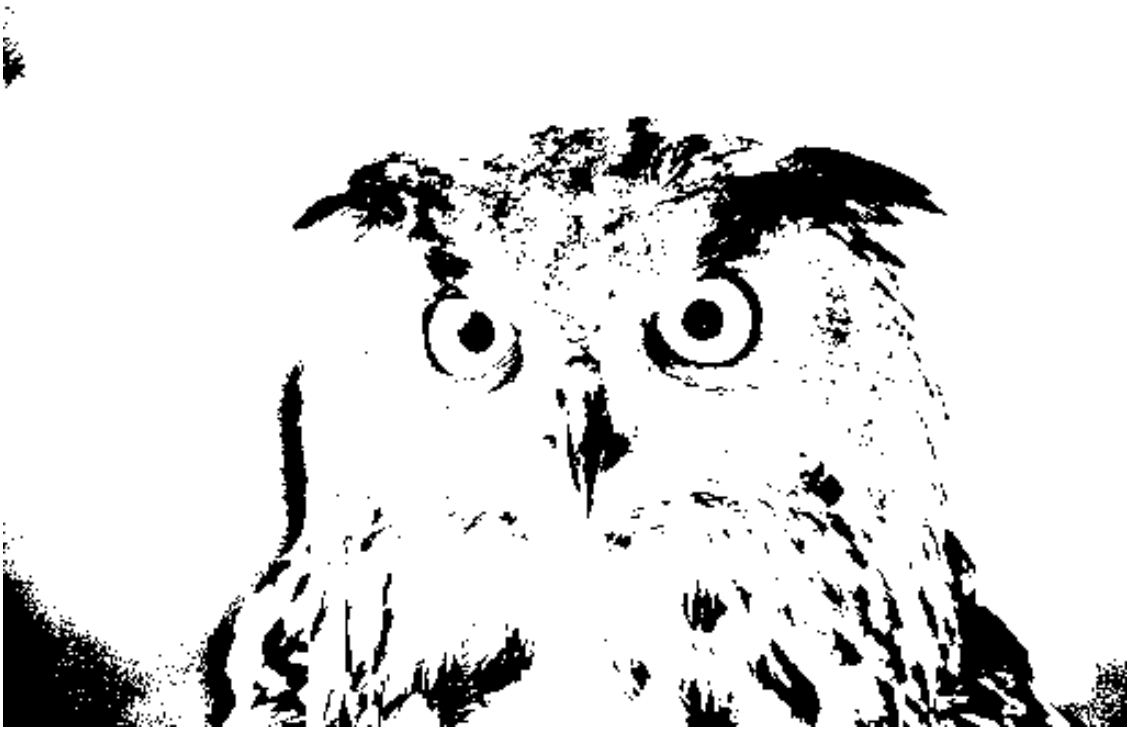
12.1 where, nonzero

We can use a Boolean array to select elements of an array with `np.where(bool, a,b)` which selects `a` where `bool` is True and `b` where `bool` is False. `bool`, `a` and `b` must be the same shape, or be broadcastable to the right shape.

[79]:
```
print(np.where(x > y, 1, 0))
```

```
[0 0 1 1]
```

[80]:
```
owl_gray = load_image_gray("imgs/owl.png")

# force to either black (0) or to white (1), by thresholding at 0.2
show_image(np.where(owl_gray<0.2, 0, 1), width="50%")
```



nonzero We can convert a boolean array to an array of **indices** with `nonzero`

[81]:
```
x = np.array([10, 7, 2, 9, 0, 5, 2, 1])
print((np.nonzero(x > 3)))
```

```
(array([0, 1, 3, 5]),)
```

[82]:
```
## find *indices* where wheat price was > 50
wheat = np.loadtxt("data/Wheat.csv", delimiter=',')
expensive_rows = np.nonzero(wheat[:,2]>50)

# use the indices to index the rows
wheat[expensive_rows]
```

[82]:
```
array([[   7.  , 1595.  ,   64.  ,    5.54],
       [  17.  , 1645.  ,   53.  ,    6.45],
       [  47.  , 1795.  ,   76.  ,   27.5 ],
       [  48.  , 1800.  ,   79.  ,   28.5 ],
```

```
       [  49.  , 1805.  ,   81.  ,   29.5 ],
       [  50.  , 1810.  ,   99.  ,   30.  ]])
```

[83]:
```
x=np.arange(5,1,-1)
print(x)
print(np.nonzero(x>2))
```

```
[5 4 3 2]
(array([0, 1, 2]),)
```

12.2 Fancy indexing

An extension of indexing, which allows us to index arrays with arrays - very powerful operator - we can select *irregular* parts of an array and perform operations on them

12.2.1 Index arrays

For example, an array of **integer** indices can be used as an index directly:

[84]:
```
x = np.array([10, 20, 30, 40])
y = np.array([0, 1, 1, 2, 2, 1])
print_matrix("x", x)
print_matrix("y", y)
print_matrix("x[y]", x[y])
```

```
x
 [[10 20 30 40]]
y
 [[0 1 1 2 2 1]]
x[y]
 [[10 20 20 30 30 20]]
```

[85]:
```
x2d = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
y = np.array([0, 1, 1])
## indexes as row slices
print_matrix('x', x2d)
print_matrix('y', y)
```

```
x
 [[10 20 30]
 [40 50 60]
 [70 80 90]]
y
 [[0 1 1]]
```

[86]:
```
print_matrix('x_{2d}[y]', x2d[y])
```

```
x_{2d}[y]
 [[10 20 30]
```

```
 [40 50 60]
 [40 50 60]]
```

Dimensions are always still separated by commas:

[87]:
```
z2d = np.array([0,0,2])
## indexes as row slices
## TRICKY: we index first by y2d (to get the array above), *then* by z2d to␣
 ↪select the columns
print((x2d[y, z2d]))
```

```
[10 40 60]
```

[88]:
```
## find indices where wheat price was > 50
expensive_indices = np.nonzero(wheat[:,2]>50)

# print the matching years
print((wheat[:,1][expensive_indices]))
```

```
[1595. 1645. 1795. 1800. 1805. 1810.]
```

12.2.2 Boolean indexing

As well as using indices, we can directly index arrays with boolean arrays.

For example, if we have an array

`x = [1,2,3]`

and an array

`bool = [True, False, True]`

then `x[bool]` is the array:

`[1,3]`

Note that this is pulling out *irregular* parts of the array (although the result is always guaranteed to be a rectangular array).

[89]:
```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8])
y = np.array([1, 0, 1, 1, 0, 0, 1, 1])
print_matrix("x", x)
print_matrix("y", y)
```

```
x
 [[1 2 3 4 5 6 7 8]]
y
 [[1 0 1 1 0 0 1 1]]
```

[90]:
```
print_matrix("x[y==0]", x[y == 0])
print_matrix("y==0", y == 0)
```

```
x[y==0]
 [[2 5 6]]
y==0
 [[False  True False False  True  True False False]]
```

[91]:
```
# boolean arrays in broadcasting
x2d = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])

y = np.array([10, 20, 100])
first_column_bigger = x2d[:, 0] <= y

print_matrix("x=", x2d)
print_matrix("y=", y)

print_matrix("\nx[x[:,0]\leq y]=", x2d[first_column_bigger])
# select the rows where the first column of x is <= correspond element of y
```

```
x=
 [[10 20 30]
 [40 50 60]
 [70 80 90]]
y=
 [[ 10  20 100]]

x[x[:,0]\leq y]=
 [[10 20 30]
 [70 80 90]]
```

[92]:
```
# the wheat example, selecting years when the cost of wheat was > 50 shillings /
  ↪ quarter bushel
print((wheat[:,1][wheat[:,2]>50])) # note the double indexing
```

```
[1595. 1645. 1795. 1800. 1805. 1810.]
```

[93]:
```
owl_masked = np.array(owl_gray) # copy the owl
# we can use fancy indexing in assignments, exactly as we would in any other␣
  ↪operation
owl_masked[owl_masked<0.5] *= 0.6 # make the dark bits darker
owl_masked[owl_masked>0.5] *= 1.6 # and the light bits lighter

show_image(owl_masked, width="50%")
```

12.3 Selection review

- **Slicing** can chop out rectangular sections of an array, including with regular gaps. `x[2:5, :]` selects rows 3-6 of `x`. `x[:, :, ::-1]` selects all of `x`, but with the last axis reversed (e.g RGB colours -> BGR colours).
- **Specific indexing** We can index anywhere we could specify a slice range with a list instead. This allows quick tricks to rearrange elements. `x[[0, 2, 1], :]` will return an array with the first, third and second rows of `x`, in that order.
- **Boolean indexing** or **masking**. if we index an array with a Boolean array instead of a slice range, we will get all of the elements where that array was True. This is particularly useful in assignments. We can, for example, write `x[x > 5] = 0` to set all value of `x` that are greater than 5 to 0.

12.4 Key Resources

- A visual guide to NumPy This is an excellent intro that covers most of the concepts here.
- NumPy tutorial

See the **Resources** page on Moodle for a full list of resources, including some more advanced material for those who are interested.

---

13 NumPy reference

This is a quick reference guide. **We won't cover this in the lecture.**

13.0.1 NumPy reference

In DSF we will use the following NumPy functions/functionality. You should know what these functions do and be ready to use them by the end of the first week.

These are all covered in the NumPy API reference

- multidimensional slicing syntax `x[1,2:5]`
- broadcasting arithmetic `x+4`
- slice assignment `x[0:5] = 1`
- boolean indexing `x[[True, False, False]]`
- fancy indexing `x[y] += 1`

13.0.2 Generation

- `np.loadtxt` / `np.savetxt`
- `np.zeros` / `np.ones` / `np.full` / `np.empty`
- `np.zeros_like` / `np.zeros_like` / `np.full_like` / `np.empty_like`
- `np.arange`
- `np.linspace`
- `np.array`
- `np.meshgrid`

13.0.3 Logical

- `np.logical_and` / `np.logical_or` / `np.logical_not`

13.0.4 Reductions

- `np.any` / `np.all`
- `np.prod`
- `np.min` / `np.max`
- `np.sum` / `np.mean` / `np.std`

13.0.5 Accumulations

- `np.cumprod` / `np.cumsum`
- `np.diff` / `np.gradient`

13.0.6 Random

- `np.random.uniform` / `np.random.normal` / `np.random.randint`
- `np.random.choice` / `np.random.permuation`

13.0.7 Reshaping

- `np.tile`

- `np.transpose` / `x.T`
- `np.stack` / `np.concatenate`
- `np.squeeze`
- `np.reshape`
- `np.einsum`
- `np.ravel`
- `np.swapaxes` / `np.rollaxes`

13.0.8 Floating point handling

- `np.isinf` / `np.isnan` / `np.isfinite`
- `np.nan` / `np.inf`
- `np.allclose`
- `np.frexp`

13.0.9 Index finding and sorting

- `np.argmin` / `np.argmax`
- `np.argsort` / `np.sort`
- `np.nonzero`
- `np.where`

13.0.10 Matrix operations

- `np.dot` / `np.inner` Inner of vectors / matrices / tensors
- `np.outer` Outer product

13.0.11 Standard functions (ufuncs)

- `np.minimum`, `np.maximum` (elementwise min/max)
- `np.add`, `np.subtract`, `np.multiply` (function forms of + and -)
- `np.log`, `np.exp`
- `np.sin`, `np.cos`, `np.tan`, `np.arcsin`, `np.arccos`, `np.tanh`, etc.