UWE Bristol University of the West of England
The British College

Fundamentals of Data Science

WEEK 7: Mathematical Computation Using NumPy

Prepared by: Krishna Devkota

The NumPy array object

What are NumPy and NumPy arrays?

- High-level number objects: integers and floating point
- Containers: lists (costless insertion and append)

- Numpy provides extension package to Python for multi-dimensional arrays
- Closer to hardware (efficiency)
- Designed for scientific computation (convenience)
- Also known as array oriented computing

In [1]:
```python
import numpy as np

print(np.__version__)
```

```
1.20.3
```

In [2]:
```python
import numpy as np # recommended import conventions

a = np.array([0,1,2,3])

type(a)
```

Out[2]:
```
numpy.ndarray
```

Why is NumPy useful?

It is a memory-efficient container that provides fast numerical operations.

Interactive Help

In [3]:
```python
np.array?
```

In [4]:
```python
np.lookfor("dot")
```

```
Search results for 'dot'
------------------------
numpy.dot
    Dot product of two arrays. Specifically,
numpy.vdot
    Return the dot product of two vectors.
numpy.tensordot
    Compute tensor dot product along specified axes.
numpy.ma.dot
    Return the dot product of two arrays.
numpy.chararray.dot
    Dot product of two arrays.
numpy.linalg.multi_dot
    Compute the dot product of two or more arrays in a single function call,
numpy.random.SFC64
    BitGenerator for Chris Doty-Humphrey's Small Fast Chaotic PRNG.
numpy.linalg.tensorinv
    Compute the 'inverse' of an N-dimensional array.
numpy.ma.MaskedArray.dot
    Masked dot product of two arrays. Note that `out` and `strict` are
```

In [5]:
```
np.con*?
```

Import conventions

In [6]:
```
import numpy as np
```

Creating arrays

Manual construction of arrays

1-D:

In [7]:
```
a = np.array([0, 1, 2, 3])
a
```

Out[7]:
```
array([0, 1, 2, 3])
```

In [8]:
```
a.ndim
```

Out[8]:
```
1
```

In [9]:
```
a.shape
```

Out[9]:
```
(4,)
```

In [10]:
```
len(a)
```

Out[10]:
```
4
```

2-D, 3-D:

In [11]:
```
b = np.array([[0, 1, 2], [3, 4, 5]])
b
```

Out[11]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [12]:
```
b.ndim
```

Out[12]:
```
2
```

In [13]:
```
b.shape
```

Out[13]:
```
(2, 3)
```

In [14]:
```
len(b)
```

Out[14]:
```
2
```

In [15]:
```
c = np.array([[[1], [2]], [[3], [4]]])
c
```

Out[15]:
```
array([[[1],
        [2]],

       [[3],
        [4]]])
```

In [16]:
```
c.shape
```

Out[16]:
```
(2, 2, 1)
```

In [17]:
```
b = np.array([[[1,4], [2,7], [4,3]],[[0, 1], [3, 4],[5,6]]] )
b.shape
```

Out[17]:
```
(2, 3, 2)
```

Exercise

- Create a simple two dimensional array. First, redo the examples from above. And then create your own: how about odd numbers counting backwards on the first row, and even numbers on the second?

- Use the functions len(), numpy.shape() on these arrays. How do they relate to each other? And to the ndim attribute of the arrays?

Functions for creating arrays

In practice, we rarely enter the items one by one.

Create evenly spaced arrays:

In [18]:
```
a = np.arange(10) # 0 .. n-1
print(a)
```

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

In [19]:
```
b = np.arange(1, 9, 2) # start, end (exclusive), step
print(b)
```

```
[1 3 5 7]
```

Create array by specifying number of points:

In [20]:
```
c = np.linspace(0, 1, 5)   # start, end, num-points
print(c)
```

```
[0.   0.25 0.5  0.75 1.  ]
```

In [21]:
```
d = np.linspace(0, 1, 5, endpoint=False)
print(d)
```

```
[0.  0.2 0.4 0.6 0.8]
```

In [22]:
```
# np.linspace?
```

Some common arrays:

Arrays with ones

In [23]:
```
a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
print(a)
```

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

Arrays with zeros (Zero Matrix)

In [24]:
```
b = np.zeros((3, 3))
print(b)
```

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

Unit Matrix (Identity Matrix)

In [25]:
```
c = np.eye(3)
print(c)
```

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

Diagonal Matrix

In [26]:
```
d = np.diag(np.array([11, 27, 34, 4]))
print(d)
```

```
[[11  0  0  0]
 [ 0 27  0  0]
 [ 0  0 34  0]
 [ 0  0  0  4]]
```

Random numbers

In [27]:
```
a = np.random.rand(70)       # uniform in [0, 1]
print(a)
```

```
[0.50755507 0.0211933  0.43352176 0.44631306 0.23881999 0.83024573
 0.74476418 0.586479   0.49286785 0.48735588 0.2667407  0.6050111
 0.75354372 0.27058423 0.52230328 0.09832853 0.71363667 0.88404059
 0.56705442 0.99448158 0.17873977 0.01220009 0.45699848 0.93175194
 0.84602469 0.47332988 0.90255503 0.22599553 0.30415374 0.71499388
 0.72409148 0.01867644 0.2858131  0.58048634 0.93078663 0.3389969
 0.12008312 0.51627271 0.69920706 0.29864068 0.86160962 0.9058072
 0.76858325 0.26123164 0.9384556  0.93864246 0.74504455 0.91073504
 0.23722471 0.49496735 0.80987834 0.95456578 0.63748325 0.91084975
 0.69213675 0.04294299 0.8335869  0.36994852 0.936557   0.48305288
 0.12533161 0.96445418 0.01702583 0.67657077 0.14043997 0.15531285
 0.64955775 0.98165422 0.69480742 0.76197389]
```

In [28]:
```
b = np.random.randn(70)      # Gaussian
print(b)
```

```
[-1.08057716 -0.22380444  0.68662417 -0.73620379 -0.01506303 -0.67925245
  0.46199332 -1.12674566  2.11978867  1.61926528  0.6148901  -0.88718431
 -1.06112917  0.86899734  0.0660828   0.8491745  -1.8879036   0.61746113
 -1.26026979 -1.21270503  0.81937935  1.83003338  0.06174604 -0.64577448
 -0.02463203  0.06691692 -1.03999184 -0.2560261  -0.23238899 -1.00755283
 -0.96900851  0.3094744  -0.11024096  1.60897676 -0.75604294  1.02829799
 -0.02520225 -0.33355825 -0.3729897   0.17476908 -1.19253672  1.30436895
 -1.41566081  0.47923289  0.23361135 -0.12545121 -0.28385152  0.26110561
  0.82626166  0.31635129  0.11070639  1.70075073  1.07674992  0.06667967
  0.01521925 -0.03649094 -1.27846407 -0.09173462 -0.44781851 -0.29782149
 -0.5062295  -0.1050317  -0.50717154  0.50516987  0.85546728  0.18512002
  1.1571755  -1.07224293  1.34049442  1.41773681]
```

Seeding a random number generator

In [29]:
```
import numpy as np
np.random.seed(42)        # Setting the random seed

b = np.random.rand(3)
print(b)
```

```
[0.37454012 0.95071431 0.73199394]
```

Exercise

- Experiment with `arange` , `linspace` , `ones` , `zeros` , `eye` and `diag` .
- Create different kinds of arrays with random numbers.
- Try setting the seed before creating an array with random values.
- Look at the function `np.empty` . What does it do? When might this be useful?

Basic Data Types

You might sometimes notice that, in some instances, array elements are displayed with a trailing dot (e.g. 2. vs 2). This is due to a difference in the data-type used:

In [30]:
```python
import numpy as np
a = np.array([1, 2, 3])
print(a.dtype)
print(a)
```

```
int32
[1 2 3]
```

In [31]:
```python
b = np.array([1., 2., 3.])
print(b.dtype)
print(b)
```

```
float64
[1. 2. 3.]
```

Different data-types allow us to store data more compactly in memory, but most of the time we simply work with floating point numbers. Note that, in the example above, NumPy auto-detects the data-type from the input.

You can explicitly specify which data-type you want:

In [32]:
```python
c = np.array([1, 2, 3], dtype=float)
print(c.dtype)
print(c)
```

```
float64
[1. 2. 3.]
```

The **default** data type is floating point:

In [33]:
```python
a = np.ones((3, 3))
print(a.dtype)
```

```
float64
```

Other Data Types

Complex:

In [34]:
```python
d = np.array([1+2j, 3+4j, 5+6*1j])
print(d.dtype)
print(d)
```

```
complex128
[1.+2.j 3.+4.j 5.+6.j]
```

Bool:

In [35]:
```python
e = np.array([True, False, False, True])
print(e.dtype)
print(e)
```

```
bool
[ True False False  True]
```

Strings:

In [36]:
```python
f = np.array(['Bonjour', 'Hello', 'Hallo'])
print(f.dtype)# <--- strings containing max. 7 letters
print(f)
```

```
<U7
['Bonjour' 'Hello' 'Hallo']
```

Others:

- int32
- int64
- uint32
- uint64

Basic visualization

We will cover visualization in detail in the coming session, but let us look at some of the basics below:

In [37]:

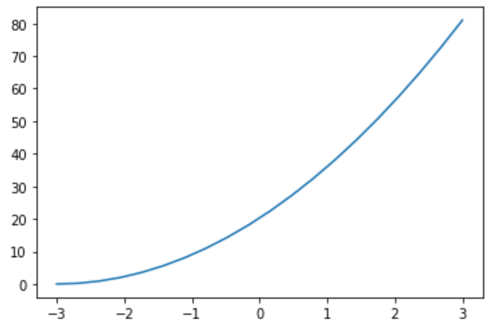
```
%matplotlib inline
import matplotlib.pyplot as plt
```

In [38]:

```
x = np.linspace(-3, 3, 20)
y = np.linspace(0, 9, 20)
plt.plot(x, y**2)       # line plot
```

Out[38]:

```
[<matplotlib.lines.Line2D at 0x1e4466c3a00>]
```

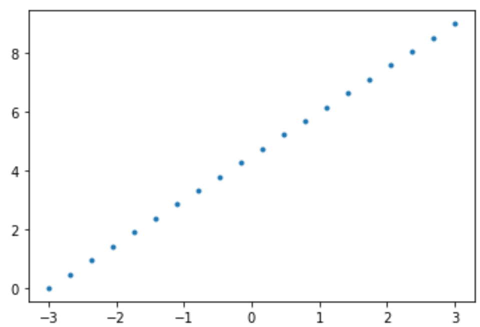


In [39]:

```
plt.plot(x, y, '.')  # dot plot
```

Out[39]:

```
[<matplotlib.lines.Line2D at 0x1e446e581c0>]
```



Plotting 2D arrays

In [40]:

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

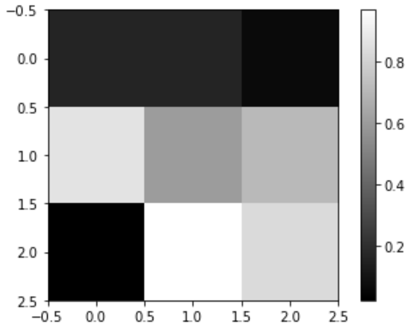
In [41]:

```
image = np.random.rand(1)
print(image)
```

```
[0.59865848]
```

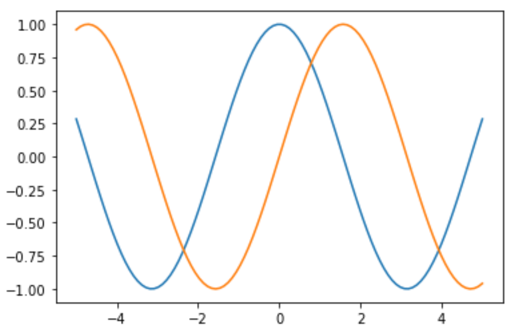
In [42]:
```python
image = np.random.rand(3,3)
print(image.shape)
print(image)
plt.imshow(image, cmap=plt.cm.gray)
plt.colorbar();
```

```
(3, 3)
[[0.15601864 0.15599452 0.05808361]
 [0.86617615 0.60111501 0.70807258]
 [0.02058449 0.96990985 0.83244264]]
```



In [43]:
```python
# Extra
a = np.linspace(-5,5,100)
plt.plot(a, np.cos(a))
plt.plot(a, np.sin(a))
```

Out[43]:
```
[<matplotlib.lines.Line2D at 0x1e446f75940>]
```



Exercise: Simple visualizations

- Plot some simple arrays: a cosine as a function of time and a 2D matrix.
- Try using the `copper` colormap on the 2D matrix.

Indexing and slicing

The items of an array can be accessed and assigned to the same way as other Python sequences (e.g. lists):

In [44]:
```python
a = np.arange(10)
print(a)
```

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

In [45]:
```python
a[0], a[2], a[-1]
```

Out[45]:
```
(0, 2, 9)
```

The usual python idiom for reversing a sequence is supported:

In [46]:
```python
a[::-1]
```

Out[46]:
```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

For multidimensional arrays, indexes are tuples of integers:

In [47]:
```python
a = np.diag([7,9,6])
a
```
Out[47]:
```
array([[7, 0, 0],
       [0, 9, 0],
       [0, 0, 6]])
```

In [48]:
```python
a[1,1]
```
Out[48]:
```
9
```

In [49]:
```python
a[2, 1] = 10 # third line, second column
a
```
Out[49]:
```
array([[ 7,  0,  0],
       [ 0,  9,  0],
       [ 0, 10,  6]])
```

In [50]:
```python
a[1]
```
Out[50]:
```
array([0, 9, 0])
```

Note:

- In 2D, the first dimension corresponds to rows, the second to columns.
- for multidimensional a, `a[0]` is interpreted by taking all elements in the unspecified dimensions.

Slicing: Arrays, like other Python sequences can also be sliced:

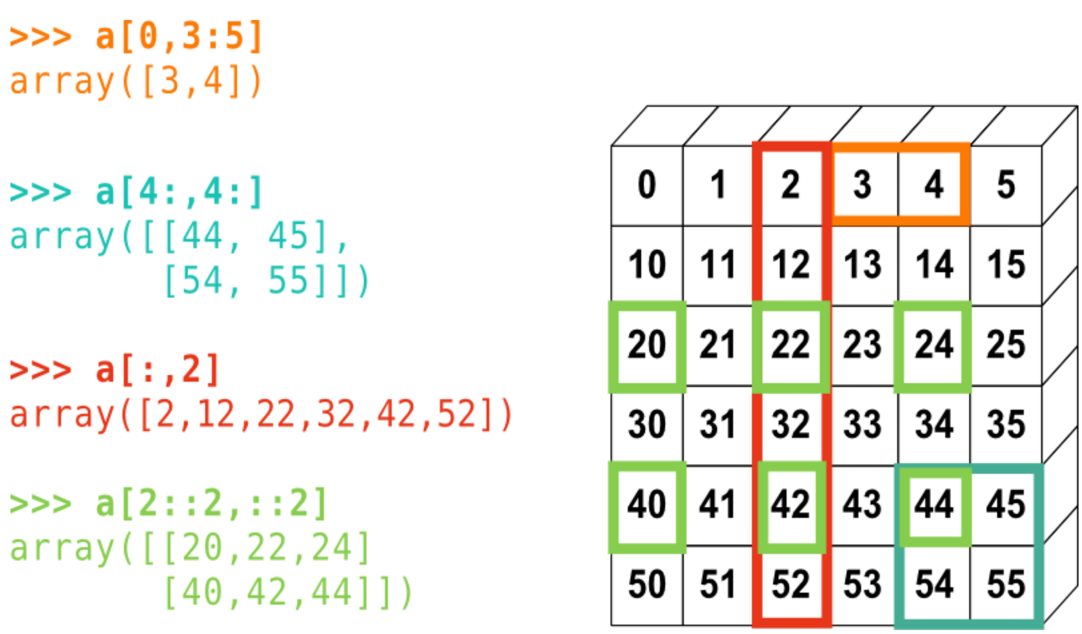
In [51]:
```python
a = np.arange(10)
a
```
Out[51]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [52]:
```python
a[2:7:2] # [start:end:step]
```
Out[52]:
```
array([2, 4, 6])
```

In [53]:
```python
a[:4]
```
Out[53]:
```
array([0, 1, 2, 3])
```

In [54]:
```python
a[1:3]
a[::2]
# a[3:]
```
Out[54]:
```
array([0, 2, 4, 6, 8])
```

An illustrated summary of NumPy indexing and slicing:



You can also combine assignment and slicing:

In [55]:
```
a = np.arange(10)
print(a)
a[5:] = 7
print(a)
```

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

In [56]:
```
b = np.arange(5)
print(b)
a[5:] = b[::-1]
print(a)
```

```
[0 1 2 3 4]
[0 1 2 3 4 4 3 2 1 0]
```

Exercise: Indexing and Slicing

- Try the different flavours of slicing, using `start` , `end` and `step` : starting from a linspace, try to obtain odd numbers counting backwards, and even numbers counting forwards.

Copies and views

A slicing operation creates a view on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory. You can use `np.may_share_memory()` to check if two arrays share the same memory block. Note however, that this uses heuristics and may give you false positives.

In [57]:
```
a = np.arange(10)
print(a)
```

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

In [58]:
```
b = a[::2]
print(b)
```

```
[0 2 4 6 8]
```

In [59]:
```
np.may_share_memory(a, b)
```

Out[59]:
```
True
```

In [60]:
```
b[0] = 12
print(b)
```

```
[12  2  4  6  8]
```

In [61]:
```
print(a)
```

```
[12  1  2  3  4  5  6  7  8  9]
```

In [62]:
```
a = np.arange(10)
c = a[::2].copy()  # force a copy
c[0] = 12
print(a)
```

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

In [63]:
```
np.may_share_memory(a, c)
```

Out[63]:
```
False
```

This behavior can be surprising at first sight… but it allows to save both memory and time.

Numerical operations on arrays

Elementwise operations

Basic operations

With scalars:

In [64]:
```
a = np.array([1, 2, 3, 4])

print(a + 1)
print(2**a)
```

```
[2 3 4 5]
[ 2  4  8 16]
```

All arithmetic operates elementwise:

In [65]:
```
b = np.ones(4) + 1
print(a - b)

j = np.arange(5)
print(2**(j + 1) - j)
```

```
[-1.  0.  1.  2.]
[ 2  3  6 13 28]
```

These operations are of course much faster than if you did them in pure python:

In [66]:
```
a = np.arange(10000)
%timeit a + 1

l = range(10000)
%timeit [i+1 for i in l]
```

```
7.03 µs ± 306 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
694 µs ± 47.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Array multiplication is not matrix multiplication:

In [67]:
```
c = np.ones((3, 3))

print(c * c) # NOT matrix multiplication!
```

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

Note: Matrix multiplication:

In [68]:
```
print(c.dot(c))
```

```
[[3. 3. 3.]
 [3. 3. 3.]
 [3. 3. 3.]]
```

A function to sompute the dotproduct of two given arrays

In [69]:
```
def dotproduct(a,b):
    if a.shape[1]!=b.shape[0]:
        print("Array multiplication not possible for given dimensions")
    else:
        return (a.dot(b))

a=np.ones((1,2))
b=np.ones((2,3))

print(a,b)
print(dotproduct(a,b))
```

```
[[1. 1.]] [[1. 1. 1.]
 [1. 1. 1.]]
[[2. 2. 2.]]
```

Exercise: Elementwise operations

- Try simple arithmetic elementwise operations: add even elements with odd elements
- Time them against their pure python counterparts using `%timeit` .
- Generate:
  - `[2**0, 2**1, 2**2, 2**3, 2**4]`
  - `a_j = 2^(3*j) - j`

Other operations

Comparisons:

In [70]:
```
import numpy as np
```

In [71]:
```
a = np.array([1, 2, 3, 4])
b = np.array([4, 2, 2, 4])

print(a == b)
print(a > b)
```

```
[False  True False  True]
[False False  True False]
```

Array-wise comparisons:

In [72]:
```
a = np.array([1, 2, 3, 4])
b = np.array([4, 2, 2, 4])
c = np.array([1, 2, 3, 4])

print(np.array_equal(a, b))
print(np.array_equal(a, c))
```

```
False
True
```

Logical operations:

In [73]:
```
a = np.array([1, 1, 1, 0], dtype=bool)
b = np.array([1, 0, 1, 0], dtype=bool)

print(np.logical_or(a, b))
print(np.logical_and(a, b))
```

```
[ True  True  True False]
[ True False  True False]
```

Transcendental functions:

In [74]:
```
a = np.arange(1,5)

print(np.sin(a))
print(np.log(a))
print(np.exp(a))
```

```
[ 0.84147098  0.90929743  0.14112001 -0.7568025 ]
[0.         0.69314718 1.09861229 1.38629436]
[ 2.71828183  7.3890561  20.08553692 54.59815003]
```

Shape mismatches

In [75]:
```
a = np.arange(4)
print(a.shape)
a + np.array([1, 2])  # Gives an error!
```

```
(4,)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_16448/2497161380.py in <module>
      1 a = np.arange(4)
      2 print(a.shape)
----> 3 a + np.array([1, 2])  # Gives an error!

ValueError: operands could not be broadcast together with shapes (4,) (2,)
```

Transposition:

In [76]:
```
a = np.triu(np.ones((3, 3)),2)   # see help(np.triu)
a = np.tril(np.ones((3, 3)), -2)
print(a)
print()
print(a.T)
```

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

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

Note: The transposition is a view

As a result, the following code is **wrong**, and will **not make a matrix symmetric:**

In [77]:
```
a += a.T
print(a)
```

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

It will work for small arrays (because of buffering) but fail for large one, in unpredictable ways.

Basic reductions

Computing sums

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

print(np.sum(x))
print(x.sum())
print(x.cumsum())
```

```
10
10
[ 1  3  6 10]
```

Sum by rows and by columns:

In [79]:
```
x = np.array([[1, 1], [2, 2]])
x
```

Out[79]:
```
array([[1, 1],
       [2, 2]])
```

In [80]:
```
x.sum(axis=0)   # columns (first dimension)
```

Out[80]:
```
array([3, 3])
```

In [81]:
```
x[:, 0].sum(), x[:, 1].sum() # alternative manual approach
```

Out[81]:
```
(3, 3)
```

In [82]:
```
x.sum(axis=1)   # rows (second dimension)
```
Out[82]:
```
array([2, 4])
```

In [83]:
```
x[0, :].sum(), x[1, :].sum() # alternative manual approach
```
Out[83]:
```
(2, 4)
```

Same idea in higher dimensions:

In [84]:
```
x = np.random.rand(2, 2, 2)

print(x)
print(x.sum(axis=2)[0, 1])
```
```
[[[0.21233911 0.18182497]
  [0.18340451 0.30424224]]

 [[0.52475643 0.43194502]
  [0.29122914 0.61185289]]]
0.48764675281297154
```

In [85]:
```
print(x[0, 1, :].sum())  # alternative approach
```
```
0.48764675281297154
```

Other reductions

Extrema:

In [86]:
```
x = np.array([1, 3, 2])

print(x.min()) # minimum value
print(x.max()) # maximum value
print()
print(x.argmin())  # index of minimum value
print(x.argmax())  # index of maximum value
```
```
1
3

0
1
```

Logical operations:

In [87]:
```
print (np.all([True, True, False]))

print(np.any([True, True, False]))
```
```
False
True
```

Note: Can be used for array comparisons:

In [88]:
```
a = np.zeros((100, 100))

print(np.any(a != 0))
print(np.all(a == 0))

a = np.array([1, 2, 3, 2])
b = np.array([2, 2, 3, 2])
c = np.array([6, 4, 4, 5])
print(((a <= b) & (b <= c)).all())
```
```
False
True
True
```

Statistics:

In [89]:
```
x = np.array([1, 2, 3, 1])
y = np.array([[1, 2, 3], [5, 6, 1]])

print(x.mean())
print(np.median(x))

print(y)
print(np.median(y,axis=-1)) # last axis

print(x.std())          # full population standard dev.
print(x.cumsum())
```

```
1.75
1.5
[[1 2 3]
 [5 6 1]]
[2. 5.]
0.82915619758885
[1 3 6 7]
```

In [90]:
```
# np.median?  # interactive help
```

Exercise: Reductions

- What is the difference between `sum` and `cumsum` ?

Worked example: Data statistics

Data in populations.txt describes the populations of hares and lynxes (and carrots) in northern Canada during 20 years.

First view the data:

In [91]:
```
!cat data/populations.txt
```

```
# year  hare    lynx    carrot
1900    30e3    4e3     48300
1901    47.2e3  6.1e3   48200
1902    70.2e3  9.8e3   41500
1903    77.4e3  35.2e3  38200
1904    36.3e3  59.4e3  40600
1905    20.6e3  41.7e3  39800
1906    18.1e3  19e3    38600
1907    21.4e3  13e3    42300
1908    22e3    8.3e3   44500
1909    25.4e3  9.1e3   42100
1910    27.1e3  7.4e3   46000
1911    40.3e3  8e3     46800
1912    57e3    12.3e3  43800
1913    76.6e3  19.5e3  40900
1914    52.3e3  45.7e3  39400
1915    19.5e3  51.1e3  39000
1916    11.2e3  29.7e3  36700
1917    7.6e3   15.8e3  41800
1918    14.6e3  9.7e3   43300
1919    16.2e3  10.1e3  41300
1920    24.7e3  8.6e3   47300
```

Now, load the data into a NumPy array:

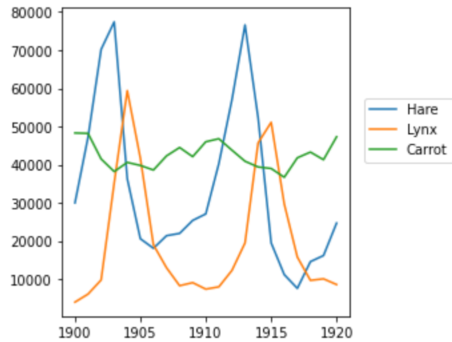
In [92]:
```
data = np.loadtxt('data/populations.txt')

year, hares, lynxes, carrots = data.T  # trick: columns to variables

year
```

Out[92]:
```
array([1900., 1901., 1902., 1903., 1904., 1905., 1906., 1907., 1908.,
       1909., 1910., 1911., 1912., 1913., 1914., 1915., 1916., 1917.,
       1918., 1919., 1920.])
```

Then, plot it:

In [93]:
```
from matplotlib import pyplot as plt
plt.axes([0.2, 0.1, 0.5, 0.8])
plt.plot(year, hares, year, lynxes, year, carrots)
plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
```

Out[93]:
```
<matplotlib.legend.Legend at 0x1e446fb5580>
```



The mean populations over time:

In [94]:
```
populations = data[:, 1:]
populations.mean(axis=0)
```

Out[94]:
```
array([34080.95238095, 20166.66666667, 42400.        ])
```

The sample standard deviations:

In [95]:
```
populations.std(axis=0)
```

Out[95]:
```
array([20897.90645809, 16254.59153691,  3322.50622558])
```

Which species has the highest population each year?:

In [96]:
```
np.argmax(populations, axis=1)
```

Out[96]:
```
array([2, 2, 0, 0, 1, 1, 2, 2, 2, 2, 2, 2, 0, 0, 0, 1, 2, 2, 2, 2, 2],
      dtype=int64)
```

Array shape manipulation

Flattening

In [97]:
```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a)
print(a.ravel())
print("\n")
print(a)
b = a.T
print(b)
print(b.ravel())
```

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


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

In [98]:
```
a.T.ravel()
```

Out[98]:
```
array([1, 4, 2, 5, 3, 6])
```

Note: In higher dimensions: last dimensions ravel out "first".

Reshaping

The inverse operation to flattening:

In [99]:
```
print(a)
```

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

In [100]:
```
print(a.ravel())
```

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

In [101]:
```
a.reshape(3,2)
print(a)
```

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

In [102]:
```
print(a.shape)
```

```
(2, 3)
```

In [103]:
```
b = a.ravel()
print(b)
print(b.shape)
b = b.reshape((3, 2))
print(b)
```

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

In [104]:
```
a.reshape((3, -1))    # unspecified (-1) value is inferred
```

Out[104]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Note: `ndarray.reshape` may return a view (cf `help(np.reshape)`)), or copy

Beware: reshape may also return a copy!

In [105]:
```
a = np.zeros((3, 2))
b = a.T.reshape(3*2)
b[0] = 9
print(a)
```

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

To understand this you need to learn more about the memory layout of a numpy array.

Adding a dimension

Indexing with the `np.newaxis` object allows us to add an axis to an array (you have seen this already above in the broadcasting section):

In [106]:
```
import numpy as np
z = np.array([1, 2, 3])
print(z)
z.shape
```

```
[1 2 3]
```

Out[106]:
```
(3,)
```

In [107]:
```python
a = z[:, np.newaxis]
print(a)
print(a.shape)
```

```
[[1]
 [2]
 [3]]
(3, 1)
```

In [108]:
```python
m = z[np.newaxis, :]
print(m)
```

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

In [109]:
```python
print(m.shape)
```

```
(1, 3)
```

Dimension shuffling

In [110]:
```python
import numpy as np
```

In [111]:
```python
a = np.arange(4*3*2).reshape(4, 3, 2)
print(a.shape)
print(a)
```

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

 [[ 6  7]
  [ 8  9]
  [10 11]]

 [[12 13]
  [14 15]
  [16 17]]

 [[18 19]
  [20 21]
  [22 23]]]
```

In [112]:
```python
print(a[2, 2, 1])
```

```
17
```

In [113]:
```python
b = a.transpose(1, 2, 0)
print(b.shape)
print(b)
```

```
(3, 2, 4)
[[[ 0  6 12 18]
  [ 1  7 13 19]]

 [[ 2  8 14 20]
  [ 3  9 15 21]]

 [[ 4 10 16 22]
  [ 5 11 17 23]]]
```

In [114]:

```
c = a.transpose(1,0,2)
print(c.shape)
print(c)
```

```
(3, 4, 2)
[[[ 0  1]
  [ 6  7]
  [12 13]
  [18 19]]

 [[ 2  3]
  [ 8  9]
  [14 15]
  [20 21]]

 [[ 4  5]
  [10 11]
  [16 17]
  [22 23]]]
```

In [115]:

```
print(b[2, 1, 0])
```

```
5
```

Also creates a view:

In [116]:

```
b[2, 1, 0] = -1
print(a[0, 2, 1])
```

```
-1
```

Resizing

Size of an array can be changed with `ndarray.resize` :

In [117]:

```
a = np.arange(4)
print(a)
a.resize((8,))
print(a)
```

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

However, it must not be referred to somewhere else:

In [118]:

```
b = a
a.resize((4,)) # Gives an error!
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_16448/2941388788.py in <module>
      1 b = a
----> 2 a.resize((4,)) # Gives an error!

ValueError: cannot resize an array that references or is referenced
by another array in this way.
Use the np.resize function or refcheck=False
```

Exercise: Shape manipulations

- Look at the docstring for `reshape` , especially the notes section which has some more information about copies and views.
- Use `flatten` as an alternative to `ravel` . What is the difference? (Hint: check which one returns a view and which a copy)
- Experiment with `transpose` for dimension shuffling.

Sorting data

Sorting along an axis:

In [119]:
```
a = np.array([[4, 3, 5], [1, 2, 1]])
print(a)
b = np.sort(a, axis=-1)
print(b)
```

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

In [120]:
```
# np.sort?  # Interactive help!
```

Note: Sorts each row separately!

In-place sort:

In [121]:
```
a.sort(axis=1)
print(a)
```

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

Finding minima and maxima:

In [122]:
```
a = np.array([4, 3, 1, 2])
j_max = np.argmax(a)
j_min = np.argmin(a)
print(j_max, j_min)
```

```
0 2
```

Exercise: Sorting

- Try both in-place and out-of-place sorting.
- Try creating arrays with different dtypes and sorting them.
- Use `all` or `array_equal` to check the results.
- Look at `np.random.shuffle` for a way to create sortable input quicker.
- Combine `ravel`, `sort` and `reshape`.
- Look at the `axis` keyword for `sort` and rewrite the previous exercise.

Summary

What do you need to know to get started?

- Know how to create arrays : `array`, `arange`, `ones`, `zeros`.
- Know the shape of the array with `array.shape`, then use slicing to obtain different views of the array: `array[::2]`, etc. Adjust the shape of the array using `reshape` or flatten it with `ravel`.
- Obtain a subset of the elements of an array and/or modify their values with masks

In [123]:
```
print(a)
```

```
[4 3 1 2]
```

In [124]:
```
a[a < 2] = 5
print(a)
```

```
[4 3 5 2]
```

- Know miscellaneous operations on arrays, such as finding the mean or max (`array.max()`, `array.mean()`). No need to retain everything, but have the reflex to search in the documentation (online docs, `help()`, `lookfor()`)!!

- For advanced use: master the indexing with arrays of integers, as well as broadcasting. Know more NumPy functions to handle various array operations.

More elaborate arrays

More data types

Casting

"Bigger" type wins in mixed-type operations:

In [125]:
```
np.array([1, 2, 3]) + 1.0
```

Out[125]:
```
array([2., 3., 4.])
```

Assignment never changes the type!

In [126]:
```
a = np.array([1, 2, 3])
print(a.dtype)
print(a)
```

```
int32
[1 2 3]
```

In [127]:
```
a[0] = 1.9     # <-- float is truncated to integer
print(a)
```

```
[1 2 3]
```

Forced casts:

In [128]:
```
a = np.array([1.7, 1.2, 1.6])
b = a.astype(int)  # <-- truncates to integer
print(b)
```

```
[1 1 1]
```

Rounding:

In [129]:
```
a = np.array([1.2, 1.5, 1.6, 2.5, 3.5, 4.5])
b = np.around(a)
print(b)                    # still floating-point
```

```
[1. 2. 2. 2. 4. 4.]
```

In [130]:
```
c = np.around(a).astype(int)
print(c)
```

```
[1 2 2 2 4 4]
```

In this session, we learnt some of the key features of NumPy. However, numPy is a very large library, and has countless other features. You can find more about it in the link below!

[http://www.scipy-lectures.org/intro/numpy/elaborate_arrays.html (http://www.scipy-lectures.org/intro/numpy/elaborate_arrays.html)](http://www.scipy-lectures.org/intro/numpy/elaborate_arrays.html)