Forward School

Program Code: J620-002-4:2020

Program Name: FRONT-END SOFTWARE DEVELOPMENT

Title : Numpy

Name: Chuay Xiang Ze

IC Number: 021224070255

Date : 22/06/2023

Introduction : Learning about numpy functions.

Conclusion : Learnt about using things like replace, random and many more.

In [1]:

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

Module P06 - Numpy

Datasets can include collections of documents, images, sound clips, numerical measurements, or, really anything. Despite the heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

Data type	Arrays of Numbers?
Images	Pixel brightness across different channels
Videos	Pixels brightness across different channels for each frame
Sound	Intensity over time
Numbers	No need for transformation
Tables	Mapping from strings to numbers

Therefore, the efficient storage and manipulation of large arrays of numbers is really fundamental to the process of doing data science. Numpy and pandas are the libraries within the SciPy stack that specialize in handling numerical arrays and data tables.

[Numpy (http://www.numpy.org/)](http://www.numpy.org/) is short for *numerical python*, and provides functions that are especially useful when you have to work with large arrays and matrices of numeric data, like matrix multiplications.

The array object class is the foundation of Numpy, and Numpy arrays are like lists in Python, except that every thing inside an array must be of the same type, like int or float. As a result, arrays provide much more efficient storage and data operations, especially as the arrays grow larger in size. However, in other ways, NumPy arrays are very similar to Python's built-in list type, but with the exception of Vectorization

Creating arrays

In [2]:

```
# Create array from lists:
lis = [[1,2,3,4,5],[6,7,8,9,10]]

ary = np.array(lis)

print(ary, type(ary))
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]] <class 'numpy.ndarray'>
```

Using array-generating functions

For larger arrays it is inpractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in numpy that generate arrays of different forms. Some of the more common are:

zeros and ones

In [3]:

```
# We use these when the elements of the
# array are originally unknown but its size is known.

np.zeros((3,4))
```

Out[3]:

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [4]:

```
np.ones((2,3,4), dtype = np.int16)
```

Out[4]:

```
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]]], dtype=int16)
```

In [5]:

```
# Create an uninitialized array of integers
# The values will be whatever happens to already exist at that memory location
np.empty((2,3))
```

Out[5]:

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

In [6]:

```
np.ones((2,3))
```

Out[6]:

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

In [7]:

```
np.zeros((2,3))
```

Out[7]:

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

In [8]:

```
np.eye(3, dtype = np.int16)
```

Out[8]:

```
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]], dtype=int16)
```

In [9]:

```
# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
```

Out[9]:

```
array([[3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

arange

In [10]:

```
# Large operations work too, and very quickly
np.arange(10000)
```

Out[10]:

```
array([   0,    1,    2, ..., 9997, 9998, 9999])
```

In [11]:

```
# reshape the 1-D array into a 2-D array
np.arange(100).reshape(10,10)
```

Out[11]:

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

random data

In [12]:

```
# Create a 3x3 array of uniformly distributed
# random values between 0 and 1
np.random.random((3, 3))
```

Out[12]:

```
array([[0.28108514, 0.18710526, 0.9639854 ],
       [0.59668895, 0.99631308, 0.53150584],
       [0.82620659, 0.33418963, 0.67780147]])
```

In [13]:

```
# Create a 3x3 array of normally distributed random values
# with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
```

Out[13]:

```
array([[-0.40910106, -0.228818  , -0.59226249],
       [-1.3704682 , -0.01518705, -1.19310839],
       [-1.9247604 ,  0.2070794 , -0.84998286]])
```

In [14]:

```
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```

Out[14]:

```
array([[7, 0, 7],
       [4, 4, 0],
       [3, 0, 6]])
```

In [15]:

```
# Create a 3x3 identity matrix
np.eye(3)
```

Out[15]:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [16]:

```
# another similar functions
np.identity(3)
```

Out[16]:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

NumPy has many functions that perform the same thing, or *can* do the same thing if used in a certain way. It's usually up to the programmer, depending on his/her familiarity with the functions, or some other specific purpose of using it (efficiency, robustness, etc.).

linspace, logspace

In [17]:

```
# Make several equally spaced points in linear space
# linspace( start, end, number of samples)
#np.linspace(0,np.pi,3)
np.linspace(0,100,11)
```

Out[17]:

```
array([  0.,  10.,  20.,  30.,  40.,  50.,  60.,  70.,  80.,  90., 100.])
```

In [18]:

```
np.logspace(0, 10, 10, base=np.e)
```

Out[18]:

```
array([1.00000000e+00, 3.03773178e+00, 9.22781435e+00, 2.80316249e+01,
       8.51525577e+01, 2.58670631e+02, 7.85771994e+02, 2.38696456e+03,
       7.25095809e+03, 2.20264658e+04])
```

In [19]:

```
import math
M = np.logspace(0, 10, 10, base=np.e)
print(math.log(M[1])-math.log(M[0]))
print(math.log(M[2])-math.log(M[1]))
print(math.log(M[3])-math.log(M[2]))
# the distance after applying log is the same. This is considered equal distance in 'log
```

```
1.111111111111111
1.1111111111111114
1.1111111111111112
```

diag

In [20]:

```
# a diagonal matrix
np.diag([1,2,3])
```

Out[20]:

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

In [21]:

```
# diagonal with offset from the main diagonal
np.diag([1,2,3], k=2)
```

Out[21]:

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

Vectorization

In [22]:

```
lis = [1,2,3,4,5]
```

In [23]:

```
lis + lis
```

Out[23]:

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

Adding two lists automatically concatenates both lists into one, but if you perform this addition when their types are **NumPy arrays**, things work out differently...

In [24]:

```
# See the difference???
np_array = np.array(lis)
np_array + np_array
```

Out[24]:

```
array([ 2,  4,  6,  8, 10])
```

In [25]:

```
print(type(lis))
print(type(np_array))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```

In [26]:

```
# Doing the same using normal lists requires a loop! Definitely NumPy is likely to be mo
print([x+x for x in lis])
print([x**2 for x in lis])
```

```
[2, 4, 6, 8, 10]
[1, 4, 9, 16, 25]
```

So we call operations on NumPy arrays as **vectorized** operations. For almost all data intensive computing, we use NumPy because of this feature, and because the whole scientific and numerical Python stack is based on NumPy.

To explain it another way, in a spreadsheet, you would add an entire column to another one by writing a formula in the first cell and auto-filling the rest of the column. Numpy does things in the similar way -- allowing such commands to be performed all in one go.

In [27]:

```
array = np.array([1, 4, 5, 8], float)
print(array)
print("")

# a 2D array/Matrix, this looks just like how we create lists...
array = np.array([[1, 2, 3], [4, 5, 6]], float)
print(array)
```

```
[1. 4. 5. 8.]

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

Numpy has all of its functionality written in *compiled* code written in C, that is much faster. But this can only be the case because all of the items in a Numpy array are of the same data type!

(Explanation: Python is dynamically typed whereas C is not - this gives extra flexibility and simplicity to Python, but makes it slower as well).

In [28]:

```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

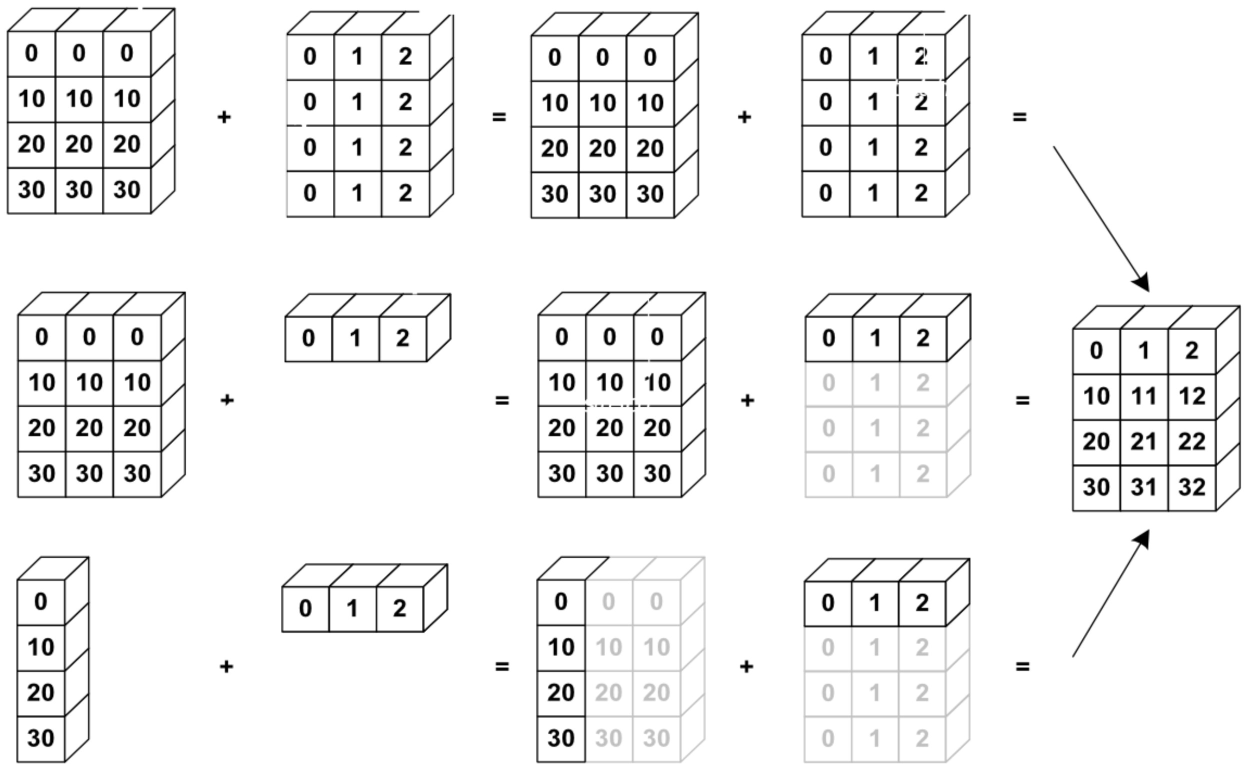
```
57.6 ms ± 1.92 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
495 µs ± 31.8 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

That's about 100 times faster.

You can index, slice, and manipulate a Numpy ***array*** the same way you would do with a Python list.

Python has a certain way of doing things. For example, the property of being "listiness". Listiness works on lists, dictionaries, files, and a general notion of something called an iterator. That's because all these objects support **the iterator protocol** - where something behaves in a list-like manner.

Broadcasting



Broadcasting is an important concept in Numpy arrays, which is simply a set of rules for applying binary ufuncs (e.g., addition, subtraction, multiplication, etc.) on arrays of different sizes. It sort of helps us to cater for operations that will be performed on array of different sizes, in an intuitive way.

In [29]:

```
M = np.ones((3, 3))
M
```

Out[29]:

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

In [30]:

```
M + 5
```

Out[30]:

```
array([[6., 6., 6.],
       [6., 6., 6.],
       [6., 6., 6.]])
```

Cool. Numpy knows that you are trying to add a single number (think of 1x1 size) to a 3x3 matrix. Mathematically, this is not possible, but Numpy knows that intuitively, you wanted to add a constant number 5 to all elements of the array.

In [31]:

```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]    # this adds a 2nd axis to array a. Basically this mea
                                    # putting the array elements along the new 2nd axis
                                    # try...
#b = np.arange(3)[np.newaxis, :]

print(a)
print(b)
print(a.shape)                     # this is only 1-D
print(b.shape[0], b.shape[1])      # the same array in 2-D representation
```

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

Of course, there are other ways of doing the same thing, that is, use the reshape function.

In [32]:

```
np.arange(3).reshape((3,1))
```

Out[32]:

```
array([[0],
       [1],
       [2]])
```

In [33]:

```
np.arange(3)[np.newaxis,np.newaxis]   # this adds 2 new axes!
```

Out[33]:

```
array([[[0, 1, 2]]])
```

In [34]:

```
print(a.shape)
print(b.shape)
a + b
```

```
(3,)
(3, 1)
```

Out[34]:

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

Now, adding a 1-D array to a 2-D array shouldn't be possible in the first place, but broadcasting allows the intuition of adding each element of one array to all elements of the other array. This resulted in a 3x3 array.

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

- **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 (left) side.
- **Rule 2:** If the shape of the 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 dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail.

In [35]:

```
# Rule one
M = np.ones((2, 3))
a = np.arange(3)
print(M)
print(a)
M + a
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
[0 1 2]
```

Out[35]:

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

In [36]:

```
# Rule two
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
print(a.shape)
print(b.shape)
print(a,b)
```

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

In [37]:

```
a + b
```

Out[37]:

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

In [4]:

```
# Rule three
M = np.ones((3, 2))
a = np.arange(3)
print(M)
print(a)
M + a
```

```
[[1. 1.]
 [1. 1.]
 [1. 1.]]
[0 1 2]
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[4], line 6
      4 print(M)
      5 print(a)
----> 6 M + a

ValueError: operands could not be broadcast together with shapes (3,2)
(3,)
```

We have a problem. Broadcasting cannot happen because the shapes are not similar and there's no way to replicate. To get over the problem, let's first ensure that they are both 2-D arrays.

In [33]:

```
# To get over the problem, add a new axis to a
print(a[:, np.newaxis].shape)
print(M.shape)
```

```
(3, 1)
(3, 2)
```

Now, it should work!

In [34]:

```
M + a[:, np.newaxis]
```

Out[34]:

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

The first array (a) is replicated along the 2nd axis and then both arrays can be added correctly.

More on broadcasting here:

[https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html (https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)](https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)

Manipulating arrays

Indexing

We can index elements in an array using square brackets and indices:

In [37]:

```
# a vector: the argument to the array function is a Python list
v = np.array([1,2,3,4])
v[0]
```

Out[37]:

```
1
```

In [39]:

```
M = np.random.random([3,3])
print(M)
# M is a 2 dimensional array, taking two indices
M[1,1]
```

```
[[0.28895494 0.94510634 0.18837395]
 [0.44883051 0.52046958 0.02201675]
 [0.11998542 0.52862201 0.5464109 ]]
```

Out[39]:

```
0.5204695816806885
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (`:`) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x` , use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0` , `stop=` *`size of dimension`* , `step=1` . We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

Source: *Python Data Science Handbook*

If we omit an index of a multidimensional array it returns the whole row (or, in general, a N-1 dimensional array)

In [40]:

```
M
```

Out[40]:

```
array([[0.28895494, 0.94510634, 0.18837395],
       [0.44883051, 0.52046958, 0.02201675],
       [0.11998542, 0.52862201, 0.5464109 ]])
```

In [41]:

```
M[1]
```

Out[41]:

```
array([0.44883051, 0.52046958, 0.02201675])
```

The same thing can be achieved with using : instead of an index:

In [42]:

```
M[1,:] #row 1
```

Out[42]:

```
array([0.44883051, 0.52046958, 0.02201675])
```

In [43]:

```
M[:,1] #column 1
```

Out[43]:

```
array([0.94510634, 0.52046958, 0.52862201])
```

We can assign new values to elements in an array using indexing:

In [44]:

```
M[0,0] = 1
```

In [45]:

```
M
```

Out[45]:

```
array([[1.        , 0.94510634, 0.18837395],
       [0.44883051, 0.52046958, 0.02201675],
       [0.11998542, 0.52862201, 0.5464109 ]])
```

In [46]:

```
# also works for rows and columns
M[1,:] = 0
M[:,2] = -1
```

In [50]:

```
M
```

Out[50]:

```
array([[ 1.        ,  0.31079608, -1.        ],
       [ 0.        ,  0.        , -1.        ],
       [ 0.55684639,  0.03668258, -1.        ]])
```

Index Slicing

Index slicing is the technical name for the syntax M[lower:upper:step] to extract part of an array:

In [74]:

```
A = np.array([1,2,3,4,5])
A
```

Out[74]:

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

In [75]:

```
A[1:3]
```

Out[75]:

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

Array slices are mutable: if they are assigned a new value the original array from which the slice was extracted is modified:

In [76]:

```
A[1:3] = [-2,-3]

A
```

Out[76]:

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

We can omit any of the three parameters in M[lower:upper:step]:

In [77]:

```
A[::] # lower, upper, step all take the default values
```

Out[77]:

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

In [78]:

```
A[::2] # step is 2, lower and upper defaults to the beginning and end of the array
```

Out[78]:

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

In [79]:

```
A[:3] # first three elements
```

Out[79]:

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

In [80]:

```
A[3:] # elements from index 3
```

Out[80]:

```
array([4, 5])
```

Index slicing works exactly the same way for multidimensional arrays:

In [27]:

```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])

A2 = np.zeros((5, 5), dtype='int16')
for m in range(5):
    for n in range(5):
        A2[m,n] = n+m*10

print(A)
print(A2)
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
```

In [82]:

```
# a block from the original array
A[1:4, 1:4]
```

Out[82]:

```
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

In [83]:

```
# strides
A[::2, ::2]
```

Out[83]:

```
array([[ 0,  2,  4],
       [20, 22, 24],
       [40, 42, 44]])
```

Fancy indexing

Fancy indexing is the name for when an array or list is used in-place of an index:

In [94]:

```
row_indices = [0, 2, 3]
A[row_indices]
```

Out[94]:

```
array([[ 0,  1,  2,  3,  4],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
```

In [103]:

```
col_indices = [1, 2, -1] # remember, index -1 means the last element
A[row_indices, col_indices]
```

Out[103]:

```
array([ 1, 22, 34])
```

In [86]:

```
A[[1],[1]]
A[1, 1]
```

Out[86]:

```
11
```

We can also use index masks: If the index mask is an Numpy array of data type bool, then an element is selected (True) or not (False) depending on the value of the index mask at the position of each element:

In [87]:

```
B = np.array([n for n in range(5)])
B
```

Out[87]:

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

In [65]:

```
row_mask = np.array([True, False, True, False, False])
B[row_mask]
```

Out[65]:

```
array([0, 2])
```

This feature is very useful to conditionally select elements from an array, using for example comparison operators:

In [9]:

```
x = np.arange(0, 10, 0.5)
x
```

Out[9]:

```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. ,
       6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
```

In [22]:

```
mask = (5 < x) * (x < 7.5)

mask
```

Out[22]:

```
array([False, False, False, False, False, False, False, False, False,
       False, False,  True,  True,  True,  True, False, False, False,
       False, False])
```

In [23]:

```
x[mask]
```

Out[23]:

```
array([5.5, 6. , 6.5, 7. ])
```

Using arrays in conditions

When using arrays in conditions,for example `if` statements and other boolean expressions, one needs to use `any` or `all` , which requires that any or all elements in the array evalutes to `True` :

In [69]:

```
M = np.array([[ 1,  4],[ 9, 16]])
M
```

Out[69]:

```
array([[ 1,  4],
       [ 9, 16]])
```

In [70]:

```
#any
if (M > 5).any():
    print("at least one element in M is larger than 5")
else:
    print("no element in M is larger than 5")

print((M>5).any())
```

```
at least one element in M is larger than 5
True
```

In [71]:

```
#all
if (M > 5).all():
    print("all elements in M are larger than 5")
else:
    print("not all elements in M are larger than 5")

print((M>5).all())
```

```
not all elements in M are larger than 5
False
```

In [72]:

```
B = np.array([n for n in range(1, 5)])
print(B)
B.mean()
```

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

Out[72]:

```
2.5
```

Functions for extracting data from arrays and creating arrays

where

The index mask can be converted to position index using the where function

In [73]:

```
indices = np.where(mask)

indices
```

Out[73]:

```
(array([11, 12, 13, 14], dtype=int64),)
```

In [74]:

```
x[indices] # this indexing is equivalent to the fancy indexing x[mask]
```

Out[74]:

```
array([5.5, 6. , 6.5, 7. ])
```

diag

With the diag function we can also extract the diagonal and subdiagonals of an array:

In [26]:

```
print(A)
np.diag(A)
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Cell In[26], line 1
----> 1 print(A)
      2 np.diag(A)

NameError: name 'A' is not defined
```

In [107]:

```
np.diag(A, -1)
```

Out[107]:

```
array([10, 21, 32, 43])
```

take

The take function is similar to fancy indexing described above:

In [77]:

```
v2 = np.arange(-3,3)
v2
```

Out[77]:

```
array([-3, -2, -1,  0,  1,  2])
```

In [78]:

```
row_indices = [1, 3, 5]
v2[row_indices] # fancy indexing
```

Out[78]:

```
array([-2,  0,  2])
```

In [79]:

```
v2.take(row_indices)
```

Out[79]:

```
array([-2,  0,  2])
```

But take also works on lists and other objects:

In [80]:

```
np.take([-3, -2, -1,  0,  1,  2], row_indices)
```

Out[80]:

```
array([-2,  0,  2])
```

choose

Constructs an array by picking elements from several arrays:

In [24]:

```
which = [1, 0, 1, 0, 2]
choices = [[-1,-2,-3,-4,-5], [1,2,3,4,5], [10, 11, 12, 13, 14]]

np.choose(which, choices)
```

Out[24]:

```
array([ 1, -2,  3, -4, 14])
```

In [25]:

```
np.choose?
```

Linear algebra

Vectorizing code is the key to writing efficient numerical calculation with Python/Numpy. That means that as much as possible of a program should be formulated in terms of matrix and vector operations, like matrix-matrix multiplication.

Scalar-array operations

We can use the usual arithmetic operators to multiply, add, subtract, and divide arrays with scalar numbers.

In [30]:

```
v1 = np.arange(0, 5)
v1
```

Out[30]:

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

In [84]:

```
v1 * 2
```

Out[84]:

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

In [85]:

```
v1 + 2
```

Out[85]:

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

In [28]:

```
print(A * 2)
print(A + 2)
```

```
[[ 0  2  4  6  8]
 [20 22 24 26 28]
 [40 42 44 46 48]
 [60 62 64 66 68]
 [80 82 84 86 88]]
[[ 2  3  4  5  6]
 [12 13 14 15 16]
 [22 23 24 25 26]
 [32 33 34 35 36]
 [42 43 44 45 46]]
```

Element-wise array-array operations

When we add, subtract, multiply and divide arrays with each other, the default behaviour is element-wise operations:

In [87]:

```
print(A)
A * A # element-wise multiplication
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
```

Out[87]:

```
array([[   0,    1,    4,    9,   16],
       [ 100,  121,  144,  169,  196],
       [ 400,  441,  484,  529,  576],
       [ 900,  961, 1024, 1089, 1156],
       [1600, 1681, 1764, 1849, 1936]])
```

In [88]:

```
print(v1)
v1 * v1
```

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

Out[88]:

```
array([ 0,  1,  4,  9, 16])
```

If we multiply arrays with compatible shapes, we get an element-wise multiplication of each row:

In [32]:

```
A.shape, v1.shape
```

Out[32]:

```
((5, 5), (5,))
```

In [90]:

```
A * v1
```

Out[90]:

```
array([[  0,   1,   4,   9,  16],
       [  0,  11,  24,  39,  56],
       [  0,  21,  44,  69,  96],
       [  0,  31,  64,  99, 136],
       [  0,  41,  84, 129, 176]])
```

Matrix algebra

What about matrix mutiplication? There are two ways. We can either use the dot function, which applies a matrix-matrix, matrix-vector, or inner vector multiplication to its two arguments:

In [33]:

```
np.dot(A,A)
```

Out[33]:

```
array([[ 300,  310,  320,  330,  340],
       [1300, 1360, 1420, 1480, 1540],
       [2300, 2410, 2520, 2630, 2740],
       [3300, 3460, 3620, 3780, 3940],
       [4300, 4510, 4720, 4930, 5140]])
```

In [111]:

```
np.dot(A, v1)
```

Out[111]:

```
array([ 30, 130, 230, 330, 430])
```

In [93]:

```
print(v1)
np.dot(v1,v1)
```

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

Out[93]:

```
30
```

Alternatively, we can cast the array objects to the type matrix. This changes the behavior of the standard arithmetic operators +, -, * to use matrix algebra.

In [94]:

```
print(A)
print(A.T)
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
[[ 0 10 20 30 40]
 [ 1 11 21 31 41]
 [ 2 12 22 32 42]
 [ 3 13 23 33 43]
 [ 4 14 24 34 44]]
```

In [35]:

```
M = np.matrix(A)
v = np.matrix(v1).T # make it a column vector
```

In [37]:

```
M
```

Out[37]:

```
matrix([[ 0,  1,  2,  3,  4],
        [10, 11, 12, 13, 14],
        [20, 21, 22, 23, 24],
        [30, 31, 32, 33, 34],
        [40, 41, 42, 43, 44]])
```

In [36]:

```
type(M)
```

Out[36]:

```
numpy.matrix
```

Notice that the type is now no longer an 'array', but a 'matrix'. This shows the "array" in matrix mode.

In [38]:

```
v
```

Out[38]:

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

In [39]:

```
M * M
```

Out[39]:

```
matrix([[ 300,  310,  320,  330,  340],
        [1300, 1360, 1420, 1480, 1540],
        [2300, 2410, 2520, 2630, 2740],
        [3300, 3460, 3620, 3780, 3940],
        [4300, 4510, 4720, 4930, 5140]])
```

In [100]:

```
M * v
```

Out[100]:

```
matrix([[ 30],
        [130],
        [230],
        [330],
        [430]])
```

If we try to add, subtract or multiply objects with incomplatible shapes we get an error:

In [101]:

```
v = np.matrix([1,2,3,4,5,6]).T
```

In [102]:

```
M.shape, v.shape
```

Out[102]:

```
((5, 5), (6, 1))
```

In [103]:

```
M * v #error due to different dimension
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
<ipython-input-103-8477489fc0d7> in <module>
----> 1 M * v #error due to different dimension

D:\Anaconda3\envs\python-dscourse\lib\site-packages\numpy\matrixlib\defmat
rix.py in __mul__(self, other)
    216         if isinstance(other, (N.ndarray, list, tuple)) :
    217             # This promotes 1-D vectors to row vectors
--> 218             return N.dot(self, asmatrix(other))
    219         if isscalar(other) or not hasattr(other, '__rmul__') :
    220             return N.dot(self, other)

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (5,5) and (6,1) not aligned: 5 (dim 1) != 6 (dim 0)
```

NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard NumPy data types are listed in the following table. Note that when constructing an array, they can be specified using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)

Data type	Description
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64` .
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128` .
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

More advanced type specification is possible, such as specifying big or little endian numbers; for more information, refer to the NumPy documentation (http://numpy.org/).

Attributes of Numpy Arrays

In [40]:

```
# Create a ranged array:
# arange = array range
a = np.arange(15)
a
```

Out[40]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Reshaping, resizing and other array properties

The shape of an Numpy array can be modified without copying the underlaying data, which makes it a fast operation even for large arrays.

In [41]:

```
# reshape it
a.reshape(3,5)
```

Out[41]:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

You can specify the type of an array when creating the Numpy array.

In [42]:

```
c = np.array([[1,2],[3,4]], dtype=complex)
c
```

Out[42]:

```
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

In [43]:

```
ndarray = np.array([[1,2,3],[4,5,6]])
print(type(ndarray))
print(ndarray)
```

```
<class 'numpy.ndarray'>
[[1 2 3]
 [4 5 6]]
```

In [44]:

```
# Number of axes or dimensions of the array (also called rank)
ndarray.ndim
```

Out[44]:

```
2
```

In [45]:

```
# Dimensions of the array:
# For a matrix with n rows and m columns,
# shape will be (n,m).
ndarray.shape
```

Out[45]:

```
(2, 3)
```

In [110]:

```
# Type of elements in the array
ndarray.dtype
```

Out[110]:

```
dtype('int32')
```

Adding a new dimension: newaxis

With newaxis, we can insert new dimensions in an array, for example converting a vector to a column or row matrix:

In [111]:

```
v = np.array([1,2,3])
```

In [112]:

```
np.shape(v)
```

Out[112]:

```
(3,)
```

In [113]:

```
# make a column matrix of the vector v
v[:, np.newaxis]
```

Out[113]:

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

In [114]:

```
# column matrix
v[:,np.newaxis].shape
```

Out[114]:

```
(3, 1)
```

In [115]:

```
v[np.newaxis,:].shape
```

Out[115]:

```
(1, 3)
```

Array Concatenation, Splitting & Stacking

In [51]:

```
# Try the following
# np.concatenate (axis = 1)
A = np.arange(10)
B = np.arange(30, 56)
print('A')
print(A)
print('B')
print(B)
np.concatenate((A, B), axis=0)
# np.split(B, 2, axis=0)
# np.hstack((A, B))
# np.vstack
np.dstack
# np.floor
# np.hsplit
# np.vsplit
# np.dsplit
```

```
A
[0 1 2 3 4 5 6 7 8 9]
B
[30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
 54 55]
```

Out[51]:

```
<function numpy.dstack(tup)>
```

In [117]:

```
# same as concatenate on axis 1
np.vstack([A, np.arange(20, 40).reshape((2,10))])
```

Out[117]:

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39]])
```

In [118]:

```
np.dstack([A,A]).shape
```

Out[118]:

```
(1, 10, 2)
```

In [128]:

```
np.hsplit?
```

Quick Exercises:

1. Create a 3x3 matrix with values ranging from 0 to 8

2. Create a 10x10 array with random values and find the minimum and maximum values
3. Create a 8x8 matrix and fill it with a checkerboard pattern
4. Create random vector of size 10 and replace the maximum value by 0
5. Create a $4 * 4$ identity matrix.
6. Generate a random $4 \times 4 \times 4$ array of Gaussianly distributed numbers.
7. Generate `n` evenly spaced intervals between 0. and 1.
8. Create a vector and then reverse the vector (first element becomes last)

Looking for more? Checkout the Neophyte, Novice, and Apprentice levels [here (http://www.loria.fr/~rougier/teaching/numpy.100/)](http://www.loria.fr/~rougier/teaching/numpy.100/).

In [46]:

```
# write your answers here or in Spyder

#1.
three_array = np.random.randint(0, 9, (3, 3))
print(three_array)

print("----------------------------------")

#2.
array = np.random.randint(0, 100, (10, 10))

minimum_value = np.min(array)
maximum_value = np.max(array)

print(array)
print("Minimum value:", minimum_value)
print("Maximum value:", maximum_value)

print("----------------------------------")

#3.
# Create an 8x8 matrix
matrix = np.zeros((8, 8), dtype=int)

# Fill the matrix with a checkerboard pattern
matrix[1::2, ::2] = 1
matrix[::2, 1::2] = 1

# Print the matrix
for row in matrix:
    print(row)

print("----------------------------------")

#4.
array2 = np.random.rand(10)

max_value = np.max(array2)
array2[array2 == max_value] = 0

print(array2)

print("----------------------------------")

#5.
identity = np.eye(4, dtype = np.int16)
print(identity)

print("----------------------------------")

#6.
normal = np.random.normal(0, 1, (4, 4))
print(normal)

print("----------------------------------")

#7.
space = np.linspace(0,1,5)
print(space)
```

```
print("---------------------------------")

#8.
array3 = np.array([1,3,5,4,2])
reverse = np.flip(array3)
print(reverse)
```

```
[[5 8 0]
 [8 7 8]
 [5 4 7]]
---------------------------------
[[ 2  5  1 10 15 49 23 17 15 99]
 [63 74  6  2 34 60 85 28 14 27]
 [38 62 30 83  8  5 30 43 26 74]
 [32 50 45 28 69 41 82 48 65 87]
 [41 20 64 38 85 44 81 84 11 48]
 [51 83 94 65 59 89 67 12  3 85]
 [24  3 29 77 77 60 13 84 33 29]
 [29 47 16 57 14 88 22 16 57 32]
 [49 21 60 39 20 47 51 85 57 46]
 [22 37 36 22 75 10 52 46 34 17]]
Minimum value: 1
Maximum value: 99
---------------------------------
[0 1 0 1 0 1 0 1]
[1 0 1 0 1 0 1 0]
[0 1 0 1 0 1 0 1]
[1 0 1 0 1 0 1 0]
[0 1 0 1 0 1 0 1]
[1 0 1 0 1 0 1 0]
[0 1 0 1 0 1 0 1]
[1 0 1 0 1 0 1 0]
---------------------------------
[0.83016547 0.65023148 0.         0.50649007 0.48951622 0.09463469
 0.23948593 0.85388782 0.24018972 0.44425594]
---------------------------------
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
---------------------------------
[[-2.35606211  0.54915926  0.39456801 -1.62660104]
 [ 1.36199212 -2.33068357  0.67694715 -0.53206991]
 [-0.68614862  0.43795475 -0.45304922 -0.5112332 ]
 [ 0.52690829 -1.30178076  2.10547814 -0.67209534]]
---------------------------------
[0.   0.25 0.5  0.75 1.  ]
---------------------------------
[2 4 5 3 1]
```

File I/O Revisited

Data processing

Often it is useful to store datasets in Numpy arrays. Numpy provides a number of functions to calculate statistics of datasets in arrays.

For example, let's calculate some properties from the Stockholm temperature dataset used above.

[http://bolin.su.se/data/stockholm/homogenized_daily_mean_temperatures.php (http://bolin.su.se/data/stockholm/homogenized_daily_mean_temperatures.php)](http://bolin.su.se/data/stockholm/homogenized_daily_mean_temperatures.php)

To download the data, click [here (http://bolin.su.se/data/stockholm/files/stockholm-historical-weather-observations-ver-1.0.2016/temperature/daily/stockholm_daily_mean_temperature_1756_2016.txt)](http://bolin.su.se/data/stockholm/files/stockholm-historical-weather-observations-ver-1.0.2016/temperature/daily/stockholm_daily_mean_temperature_1756_2016.txt)

In [41]:

```
#store data from dat format to 'data' variable
data = np.genfromtxt('./Data Files/stockholm_daily_mean_temperature_1756_2017.txt')
```

In [42]:

```
#96594 rows and 7 columns
data.shape
```

Out[42]:

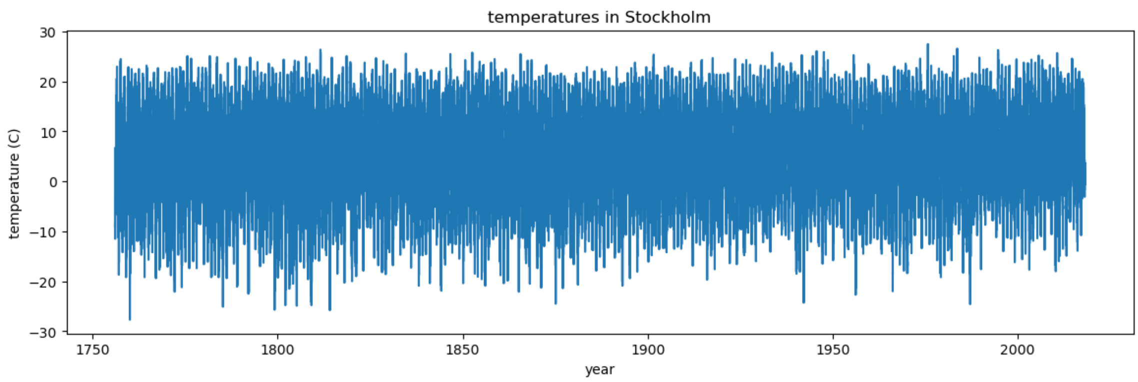
```
(95694, 7)
```

In [43]:

```
#visualize the data
print(len(data[:,0]+data[:,1]/12.0))
print(data[:,2]/365)
print(data[:,0]+data[:,1]/12.0+data[:,2]/365)

fig, ax = plt.subplots(figsize=(14,4))
ax.plot(data[:,0]+data[:,1]/12.0+data[:,2]/365, data[:,5])
ax.axis('tight')
ax.set_title('temperatures in Stockholm')
ax.set_xlabel('year')
ax.set_ylabel('temperature (C)');
```

```
95694
[0.00273973 0.00547945 0.00821918 ... 0.07945205 0.08219178 0.08493151]
[1756.08607306 1756.08881279 1756.09155251 ... 2018.07945205 2018.08219178
 2018.08493151]
```



mean

In [125]:

```
# the temperature data is in column 3
np.mean(data[:,3])
```

Out[125]:

```
6.12563065604949
```

The daily mean temperature in Stockholm since year 1756 has been about 6.11 C.

standard deviations and variance

In [126]:

```
np.std(data[:,3]), np.var(data[:,3])
```

Out[126]:

```
(12.010007951154332, 144.24029098679026)
```

min and max

In [127]:

```
# lowest daily average temperature
np.min(data[365*50:,3])
np.min(data[:,3])
```

Out[127]:

```
-999.0
```

In [128]:

```
# lowest daily average temperature
np.max(data[:,3])
```

Out[128]:

```
28.3
```

Computations on subsets of arrays

We can compute with subsets of the data in an array using indexing, fancy indexing, and the other methods of extracting data from an array (described above).

For example, let's go back to the temperature dataset:

The dataformat is: year, month, day, daily average temperature, low, high, location.

If we are interested in the average temperature only in a particular month, say April, then we can create a

In [129]:

```
np.unique(data[:,1]) # the month column takes values from 1 to 12
```

Out[129]:

```
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12.])
```

In [130]:

```
mask_april = data[:,1] == 4
```

In [131]:

```
# the temperature data is in column 3
np.mean(data[mask_april,3])
```

Out[131]:

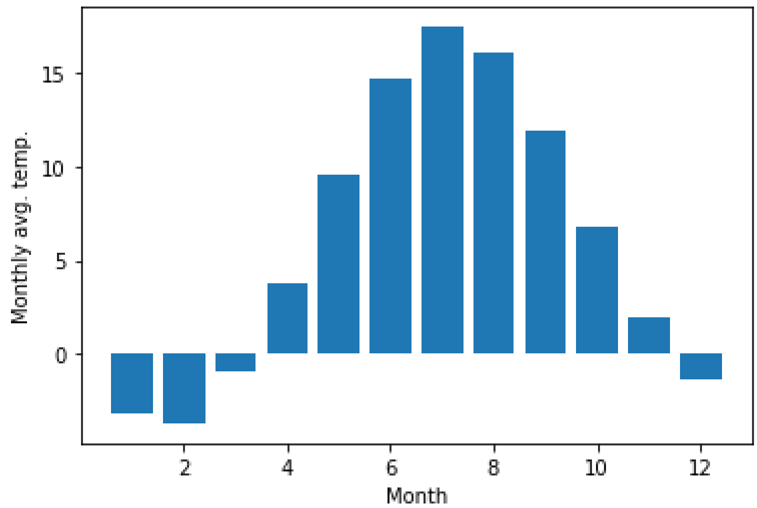
```
3.796819338422392
```

With these tools we have very powerful data processing capabilities at our disposal. For example, to extract the average monthly average temperatures for each month of the year only takes a few lines of code:

In [132]:

```
months = np.arange(1,13)
monthly_mean = [np.mean(data[data[:,1] == month, 3]) for month in months]

fig, ax = plt.subplots()
ax.bar(months, monthly_mean)
ax.set_xlabel("Month")
ax.set_ylabel("Monthly avg. temp.");
```



Calculations with higher-dimensional data

When functions such as min, max, etc. are applied to a multidimensional arrays, it is sometimes useful to apply the calculation to the entire array, and sometimes only on a row or column basis. Using the axis argument we can specify how these functions should behave:

In [133]:

```
m = np.random.rand(3,3)
m
```

Out[133]:

```
array([[0.16931816, 0.93161988, 0.05312506],
       [0.81378994, 0.80902762, 0.56730743],
       [0.38932562, 0.36546258, 0.88327177]])
```

In [134]:

```
# global max
m.max()
```

Out[134]:

```
0.9316198778199416
```

In [135]:

```
# max in each column
m.max(axis=0)
```

Out[135]:

```
array([0.81378994, 0.93161988, 0.88327177])
```

In [136]:

```
# max in each row
m.max(axis=1)
```

Out[136]:

```
array([0.93161988, 0.81378994, 0.88327177])
```

In [137]:

```
# 'out' terminology (which skips the stepp of assigning it to a temporary array)
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
```

```
[ 0. 10. 20. 30. 40.]
```

In [138]:

```
x = np.arange(1, 6)
print(np.add.reduce(x))
print(np.multiply.reduce(x))
print(np.add.accumulate(x))
print(np.multiply.accumulate(x))
```

```
15
120
[ 1  3  6 10 15]
[  1   2   6  24 120]
```

In [139]:

```
# Outer Products
x = np.arange(1, 6)
np.multiply.outer(x, x)
```

Out[139]:

```
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

Other aggregation functions

NumPy provides many other aggregation functions, but we won't discuss them in detail here. Additionally, most aggregates have a `NaN` -safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point `NaN` value (for a fuller discussion of missing data, see [Handling Missing Data (03.04-Missing-Values.ipynb)](03.04-Missing-Values.ipynb)). Some of these `NaN` -safe functions were not added until NumPy 1.8, so they will not be available in older NumPy versions.

The following table provides a list of useful aggregation functions available in NumPy:

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

Source: Python Data Science Handbook