L#13 Arrays [Functions & Processing]

October 2019, April 2020 (---que añito---)

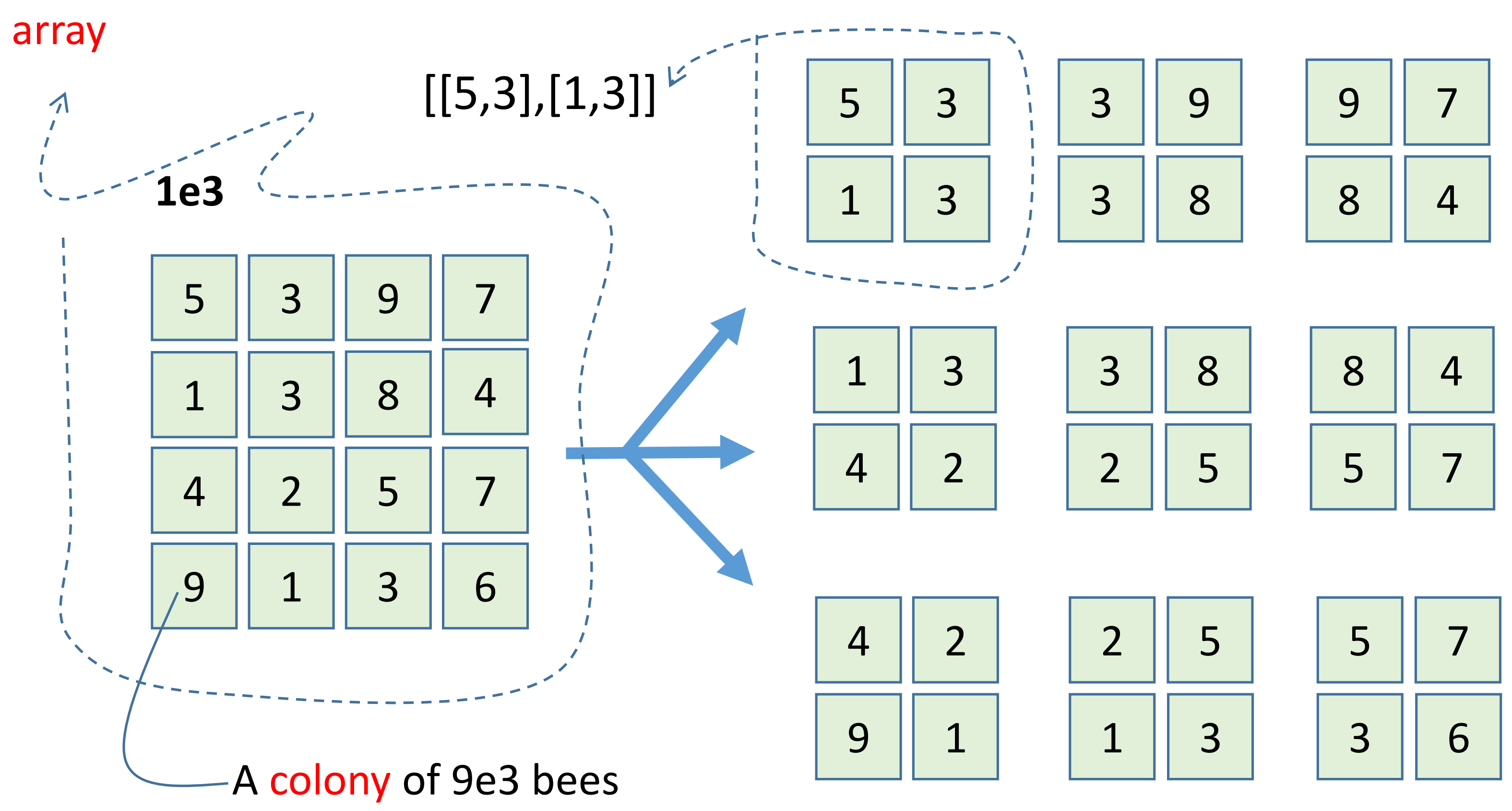
Dr. Marco A Arocha

https://docs.python.org/2.5/lib/typesseq.html

“Clusters” in 2D Arrays

Video: L13_Clusters.mp4

Clusters of 4 bee ‘overlapping’ colonies



5	3	9	7
1	3	8	4
4	2	5	7
9	1	3	6

A colony of 9e3 bees

5	3
1	3

3	9
3	8

9	7
8	4

1	3
4	2

3	8
2	5

8	4
5	7

4	2
9	1

2	5
1	3

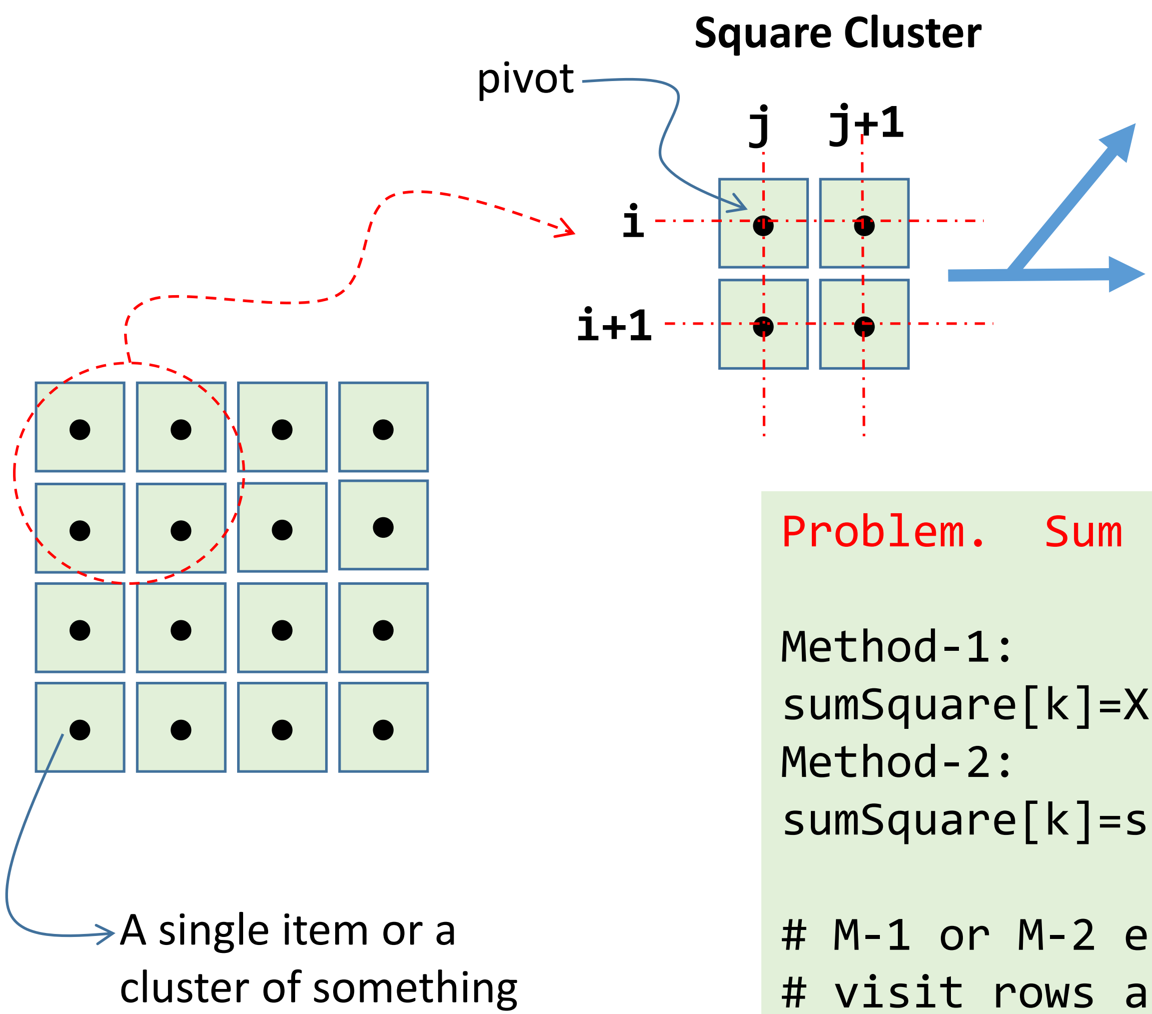
5	7
3	6

Example tasks:

- Sum
- Product
- Maximum (largest)
- Minimum (smallest)
- Average (mean)
- Even, odd, prime
- Count
- Sort (ascending, descending)
- Search
- Slicing
- Indexing (o position)
- Stacking (concatenate)
- Reshape
- Apply universal function:
- sum, sin, abs, log

Honey **bees** are social insects which live together in large, well-organized family groups. A colony typically consists of three kinds of adult **bees**: workers, drones, and a queen.

“Clusters” in 2D Arrays



A single item or a cluster of something

Method-1 (formulate):

```
X[i,j]
X[i,j+1]
X[i+1,j]
X[i+1,j+1]
```

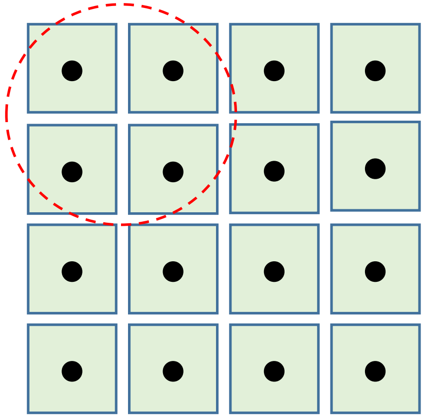
Method-2 (slicing):

```
X[i:i+2,j:j+2]
```

Problem. Sum elements in a cluster:

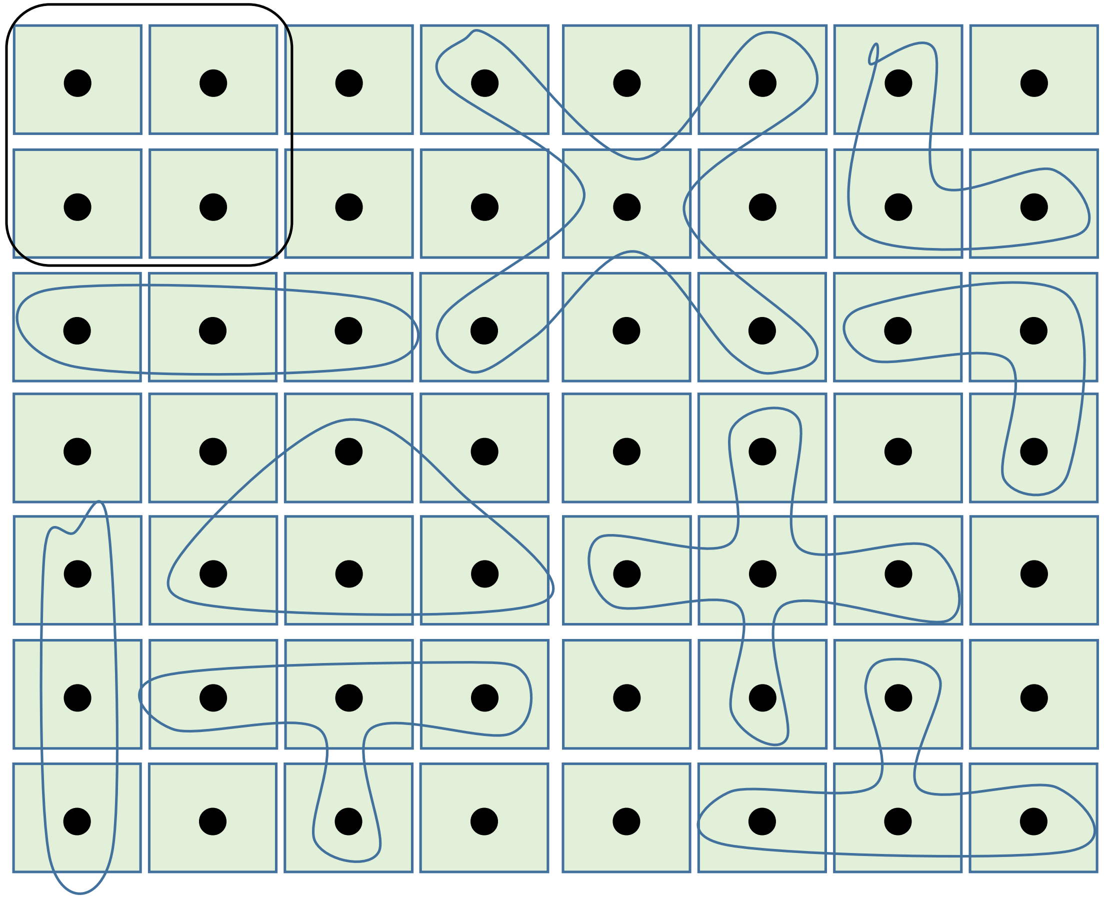
```
Method-1:
sumSquare[k]=X[i,j]+X[i,j+1]+X[i+1,j]+X[i+1,j+1]
Method-2:
sumSquare[k]=sum(X[i:i+2,j:j+2]

# M-1 or M-2 encapsulated into nested loops to
# visit rows and columns [i,j] of whole array
```

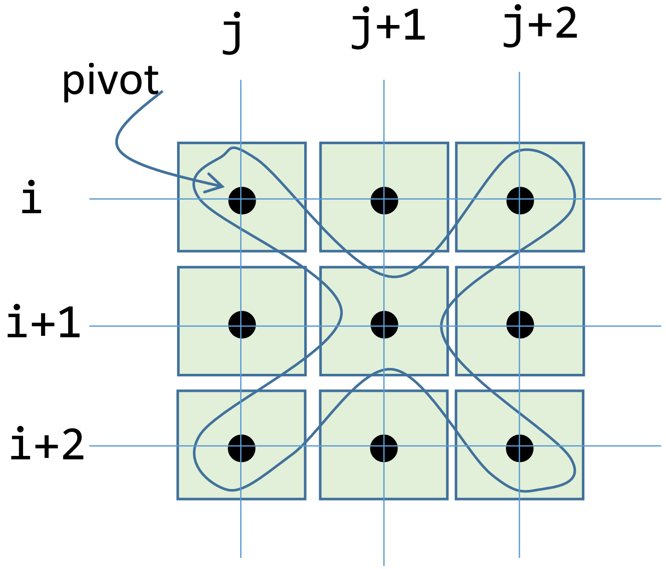
“Clusters” in 2D Arrays



```
Problem.  Sum elements of each square clusters:
# input array X, bla, bla, bla—customer style

n,m=shape(X)
N=  # how many squares fit in row direction
M=  # how many squares fit in col direction

k=   # initialize k
for i in range(N)
  for j in range(M)
     sumSquare[k]=X[i,j]+X[i,j+1]+X[i+1,j]+X[i+1,j+1]  # or
     sumSquare[k]=sum(X[i:i+2,j:j+2]
     k+=1


# Output--elaborate
print(sumSquare)
```

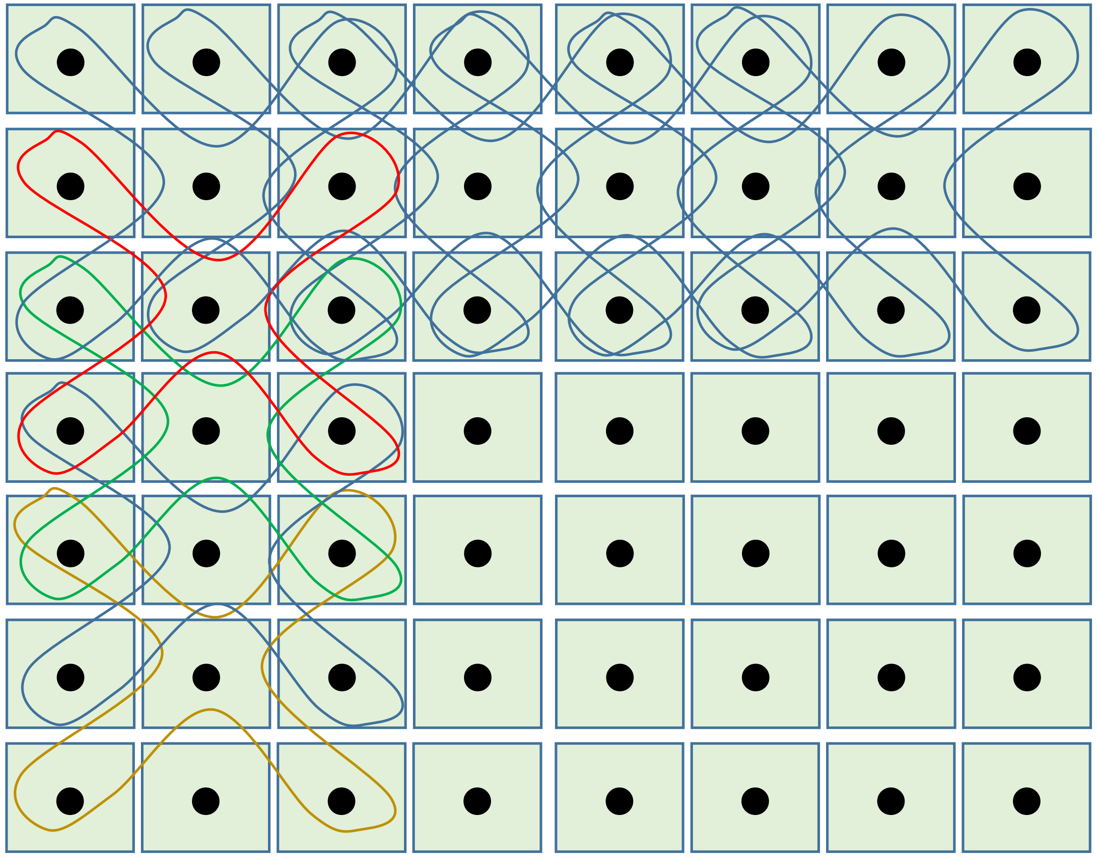
“Clusters” in 2D Arrays

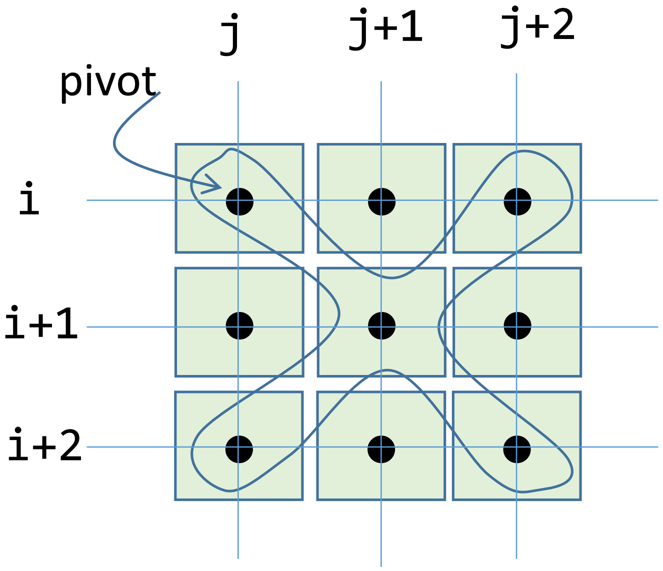




Choose a pivot element, all other elements are referenced based on pivot.

"Clusters" in 2D Arrays





- Choose a pivot element, all other elements are referenced based on pivot.
- Assume you want to add the elements in this shape.

$$CroixShape[k] = X[\,i,j\,] + X[i,j+2] + X[i+1,j+1] + X[i+2,j] + X[i+2,j+2]$$

Example: Array Element Overlap(1)

FILE: nestedLoops02.py

A	1	3	5	7	9
B	2	4	9	7	5
C					

"nada"

```
A=np.array([1,3,5,7,9],int)

B=np.array([2,4,9,7,5],int)

C=np.array([],int)  #initially empty
```

Nested Loops

```
for a in A:
    for b in B:
        if a==b:
            C=np.append(C,a)

print("C={}".format(C))
```

List Comprehension

```
C=np.array([a for a in A for b in B if a==b])
print("C={}".format(C))


# This solution doesn't need of:
# C=np.array([],int)
```

OUTPUT:
C=[5 7 9]

Example: Array Element Overlap(2)

FILE: nestedLoops02.py

A	1	3	5	7	9	
B	2	4	9	7	5	
C						“nada”

```
A=np.array([1,3,5,7,9],int)
B=np.array([2,4,9,7,5],int)
C=np.array([],int)
```

in, not in **(membership operators)**	**List Comprenhension**

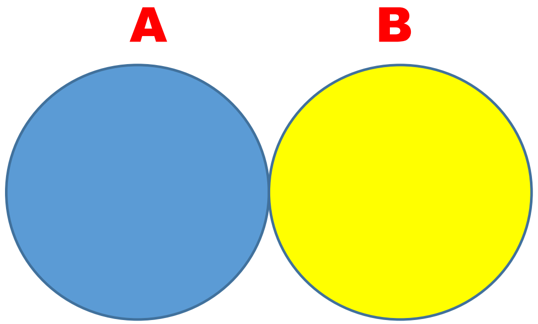
for a in A:
    if a in B:
        if a not in C:
            C=np.append(C,a)
 	
print("C={}".format(C))
	C=np.array([a for a in A if a in B if a not in C])
	print("C={}".format(C))

OUTPUT:

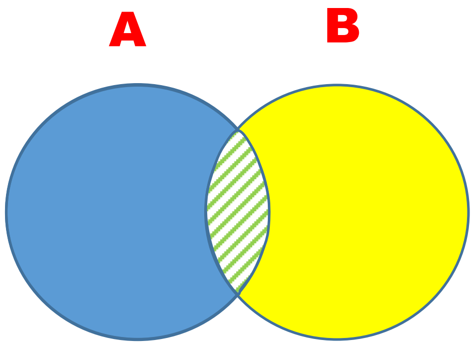
C=[5 7 9]

Venn diagram and overlapping

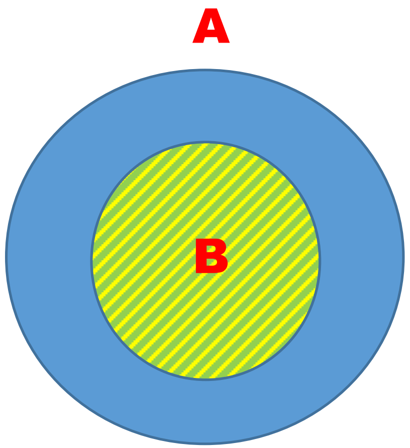
Possibilities are:



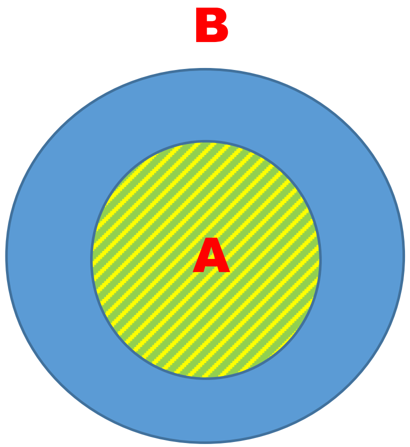
$C=A\cap B=\phi$



$C=A\cap B$



$C=A\cap B=B$



$C=A\cap B=A$

Elements that are in both sets: A and B

Creating 2D (or 3D) arrays: Reshape

arr

8	7	6	5	4	3	2	1

FILE: ArrayReshape.py

arr.reshape(2, 4)

8	7	6	5
4	3	2	1

arr.reshape(4 ,2)

8	7
6	5
4	3
2	1

arr.reshape(2, 2, 2)

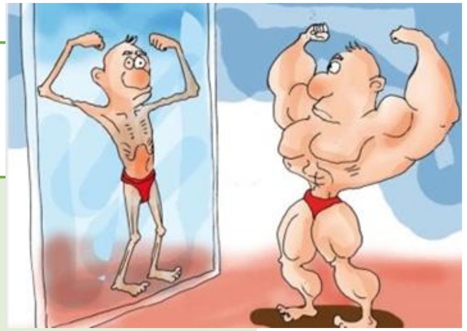
8	7
6	5

4	3
2	1

3D array

Reshape runs along the axis-0 of original array distributing elements in the new shape row by row (and page by page, if needed)

Reshape



```
import numpy as np

arr = np.arange(8,0,-1)
print("Original array : \n", arr)

# shape array with 2 rows and 4 columns
arr1 = arr.reshape(2, 4)
print("\nArray reshaped to 2 rows and 4 columns : \n", arr1)

# shape array with 2 rows and 4 columns
arr2 = arr.reshape(4 ,2)
print("\nArray reshaped to 2 rows and 4 columns : \n", arr2)

# Constructs 3D array
arr3 = arr.reshape(2, 2, 2)
print("\nOriginal array reshaped to 3D : \n", arr3)
```

Output

```
Original array :
 [8 7 6 5 4 3 2 1]

Array reshaped to 2 rows and 4 columns :
 [[8 7 6 5]
 [4 3 2 1]]

array reshaped to 2 rows and 4 columns :
 [[8 7]
 [6 5]
 [4 3]
 [2 1]]

Original array reshaped to 3D :
 [[[8 7]
  [6 5]]

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

FILE: ArrayReshape.py

Flatten Arrays

aplanar

Return a copy of the array collapsed into one dimension.

Original array: arr2

8	7	6	5
4	3	2	1

arr2=arr1.flatten(order='C')

arr2	8	7	6	5	4	3	2	1

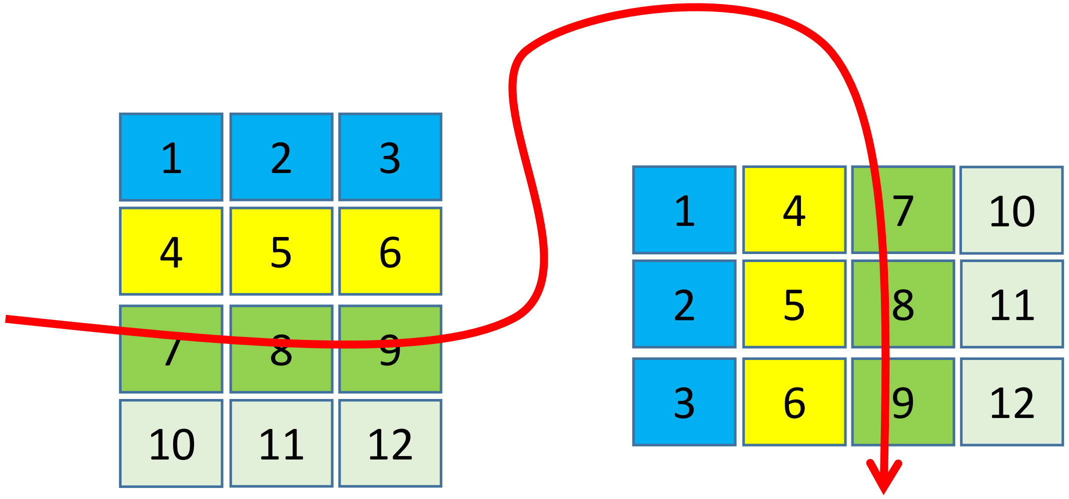
arr2=arr1.flatten(order='F')

arr2	8	4	7	3	6	2	5	1

Order options : {'C', 'F', 'A', 'K'}, order 'C' means to flatten in row-major (C-style) order 'F' means to flatten in column-major (Fortran-style) order.

FILE:flattenArray.py

Transpose: rows become columns



1	2	3
4	5	6
7	8	9
10	11	12

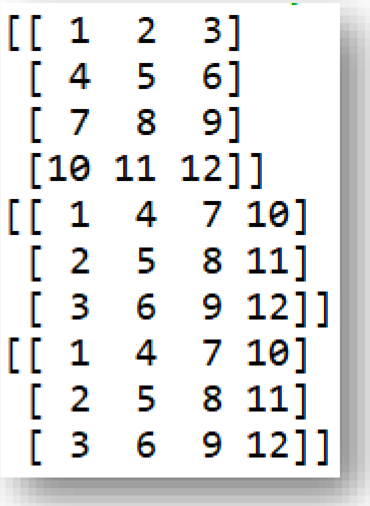
1	4	7	10
2	5	8	11
3	6	9	12

Mathematics:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

```
import numpy as np

A=np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
print(A)

AT=A.transpose()
print(AT)

AT=A.T
print(AT)
```

OUTPUT



```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[[ 1  4  7 10]
 [ 2  5  8 11]
 [ 3  6  9 12]]
[[ 1  4  7 10]
 [ 2  5  8 11]
 [ 3  6  9 12]]
```

FILE: transposeArray.py

2D array: Elementwise Operations

```
import numpy as np
```

$$\begin{bmatrix}5. & 3. & 9.\\ 3. & 9. & 5.\\ 9. & 5. & 3.\end{bmatrix} + \begin{bmatrix}1. & 1. & 1.\\ 1. & 1. & 1.\\ 1. & 1. & 1.\end{bmatrix} = \begin{bmatrix}6. & 4. & 10.\\ 4. & 10. & 6.\\ 10. & 6. & 4.\end{bmatrix}$$

```
print(np.array([ [5,3,9],[3,9,5],[9,5,3]],float)+
np.ones((3,3)))
```

$$\begin{bmatrix}5. & 3. & 9.\\ 3. & 9. & 5.\\ 9. & 5. & 3.\end{bmatrix} - \begin{bmatrix}1. & 1. & 1.\\ 1. & 1. & 1.\\ 1. & 1. & 1.\end{bmatrix} = \begin{bmatrix}4. & 2. & 8.\\ 2. & 8. & 4.\\ 8. & 4. & 2.\end{bmatrix}$$

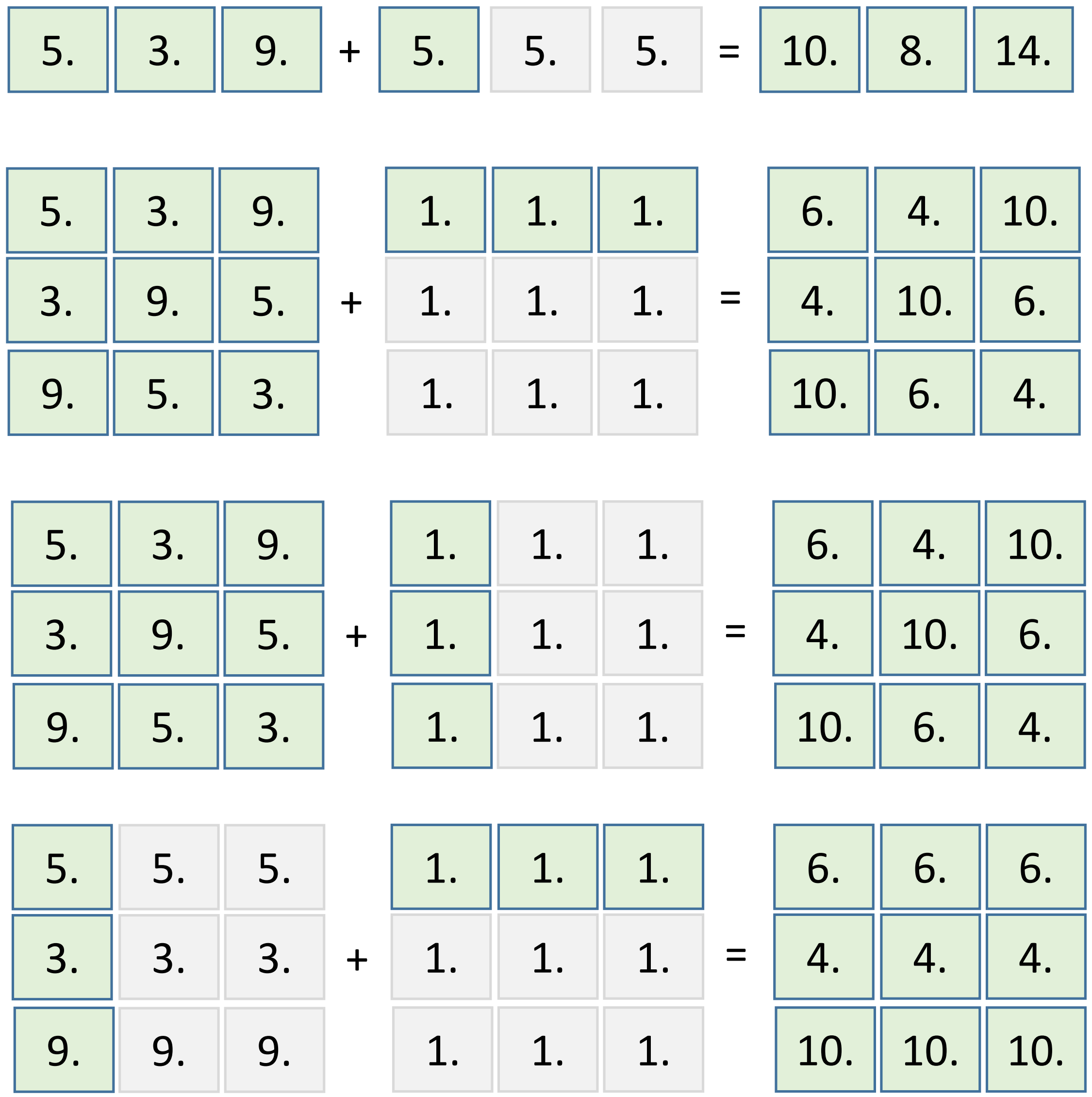
```
print(np.array([ [5,3,9],[3,9,5],[9,5,3]],float)-
np.ones((3,3)))
```

$$\begin{bmatrix}5. & 5. & 5.\\ 3. & 3. & 3.\\ 9. & 9. & 9.\end{bmatrix} * \begin{bmatrix}1. & 1. & 1.\\ 2. & 2. & 2.\\ .1 & .1 & .1\end{bmatrix} = \begin{bmatrix}5. & 5. & 5.\\ 6. & 6. & 6.\\ .9 & .9 & .9\end{bmatrix}$$

```
print(np.array([ [5,5,5],[3,3,3],[9,9,9]],float)*
[[1,1,1],[2,2,2],[.1,.1,.1]])
```

```
Operator can be: +, -, *, /, %, //
```

Broadcasting (1)



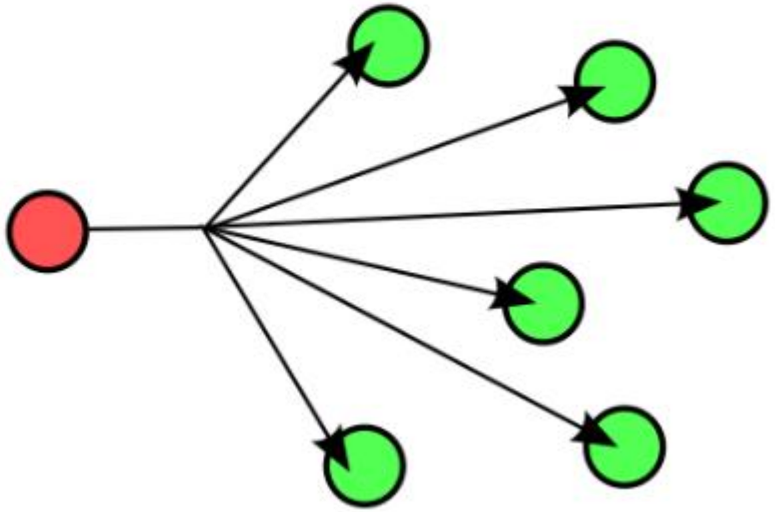
$$\begin{bmatrix} 5. & 3. & 9. \end{bmatrix} + \begin{bmatrix} 5. & 5. & 5. \end{bmatrix} = \begin{bmatrix} 10. & 8. & 14. \end{bmatrix}$$

$$\begin{bmatrix} 5. & 3. & 9. \\ 3. & 9. & 5. \\ 9. & 5. & 3. \end{bmatrix} + \begin{bmatrix} 1. & 1. & 1. \\ 1. & 1. & 1. \\ 1. & 1. & 1. \end{bmatrix} = \begin{bmatrix} 6. & 4. & 10. \\ 4. & 10. & 6. \\ 10. & 6. & 4. \end{bmatrix}$$

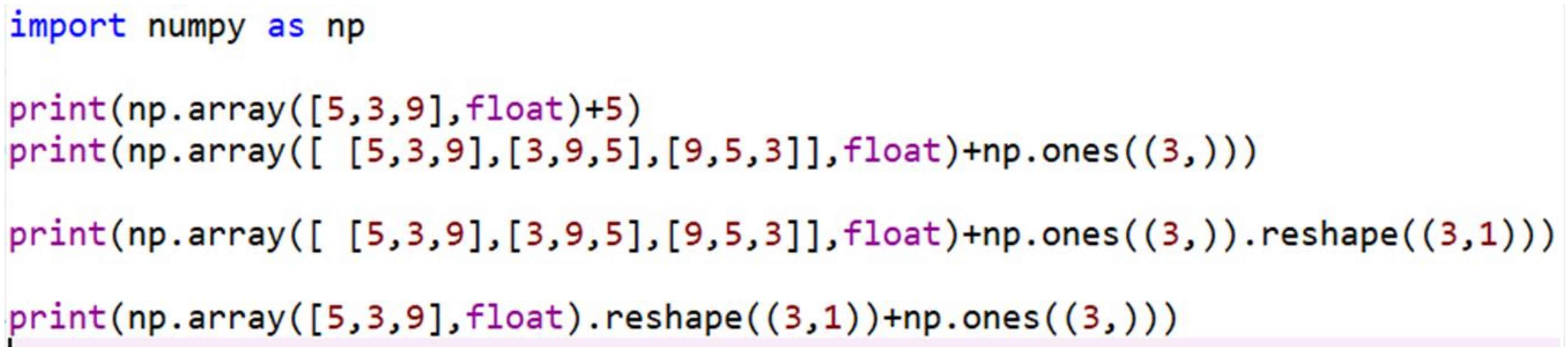
$$\begin{bmatrix} 5. & 3. & 9. \\ 3. & 9. & 5. \\ 9. & 5. & 3. \end{bmatrix} + \begin{bmatrix} 1. & 1. & 1. \\ 1. & 1. & 1. \\ 1. & 1. & 1. \end{bmatrix} = \begin{bmatrix} 6. & 4. & 10. \\ 4. & 10. & 6. \\ 10. & 6. & 4. \end{bmatrix}$$

$$\begin{bmatrix} 5. & 5. & 5. \\ 3. & 3. & 3. \\ 9. & 9. & 9. \end{bmatrix} + \begin{bmatrix} 1. & 1. & 1. \\ 1. & 1. & 1. \\ 1. & 1. & 1. \end{bmatrix} = \begin{bmatrix} 6. & 6. & 6. \\ 4. & 4. & 4. \\ 10. & 10. & 10. \end{bmatrix}$$

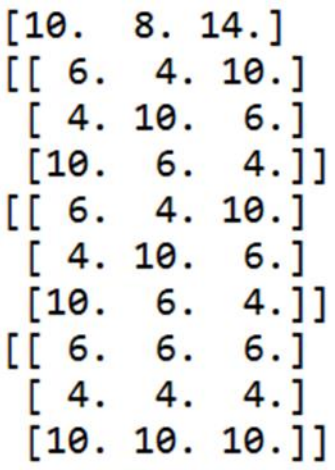
Allows to make operations with arrays of different shapes



Broadcasting(2): code & output

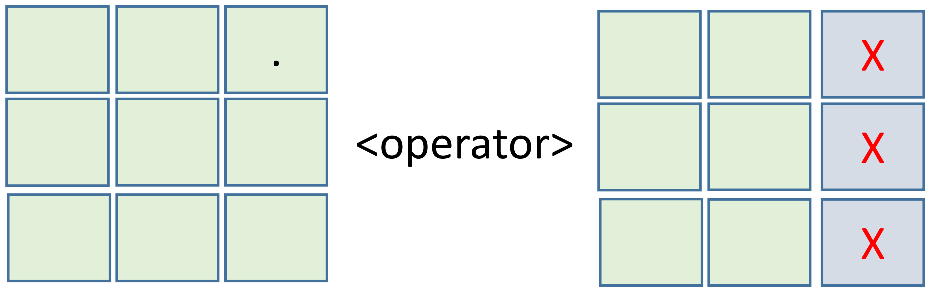


```python
import numpy as np

print(np.array([5,3,9],float)+5)
print(np.array([ [5,3,9],[3,9,5],[9,5,3]],float)+np.ones((3,)))

print(np.array([ [5,3,9],[3,9,5],[9,5,3]],float)+np.ones((3,)).reshape((3,1)))

print(np.array([5,3,9],float).reshape((3,1))+np.ones((3,)))
```



```
[10.  8. 14.]
[[ 6.  4. 10.]
 [ 4. 10.  6.]
 [10.  6.  4.]]
[[ 6.  4. 10.]
 [ 4. 10.  6.]
 [10.  6.  4.]]
[[ 6.  6.  6.]
 [ 4.  4.  4.]
 [10. 10. 10.]]
```

Cann’t broadcast sorry



Numpy Universal Functions [ufunc] : Element-Wise Operations

Numpy offers a large library of common mathematical functions called ufunc that compute element-wise on arrays, some examples are:

- abs(), sqrt()
- log(), log10()
- exp(), sin()
- cos(), tanh()
- arcsin()

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

1	2	3
4	5	6

Sine is computed for each ONE of the arguments and stored into an array of same shape as argument

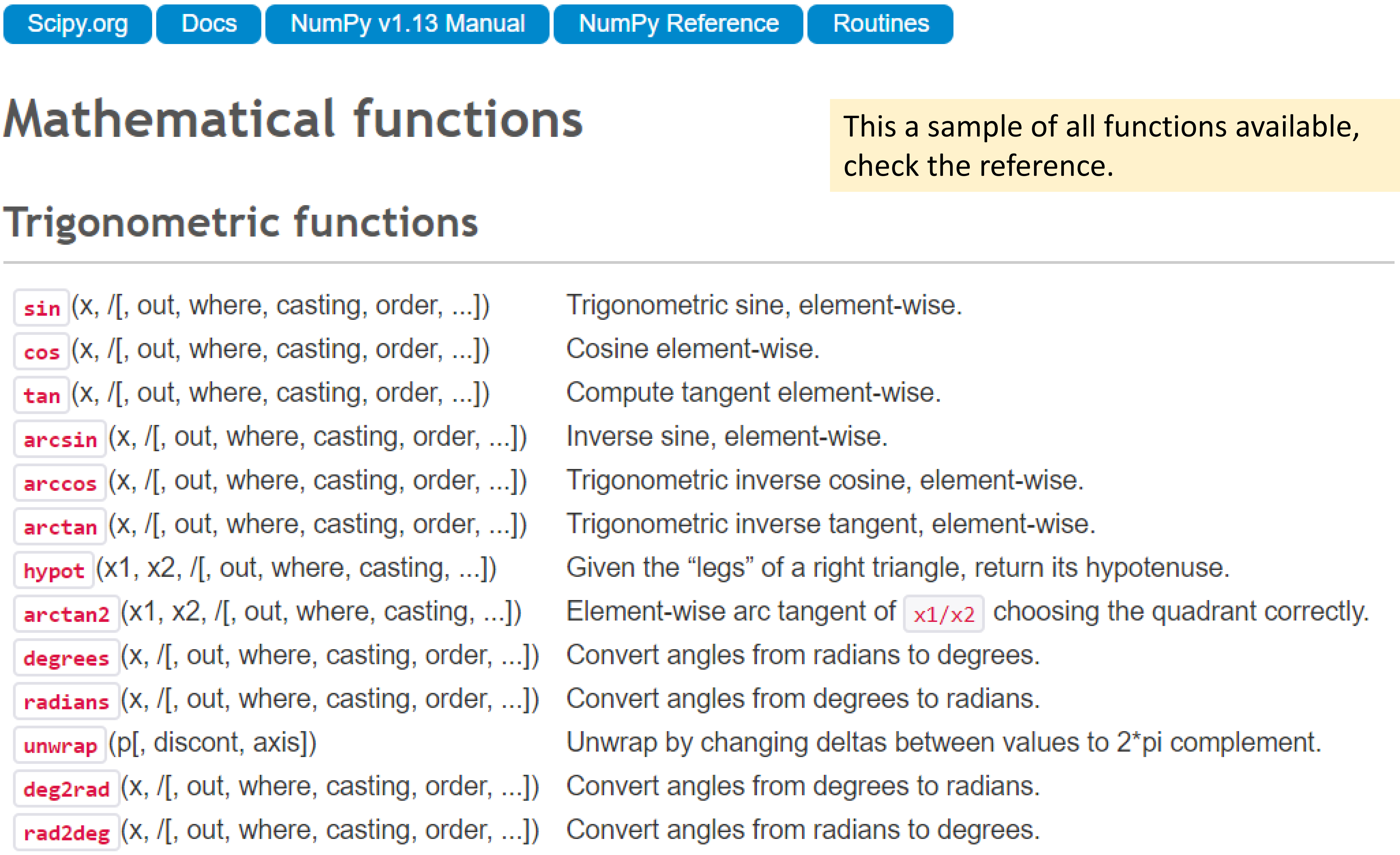
y=np.sin(x)

0.8414	0.9093	0.1411
-0.7568	-0.9589	-0.2794

```
[[ 0.84  0.91  0.14]
 [-0.76 -0.96 -0.28]]
```

Visit the site, please

https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.math.html



Mathematical functions

This a sample of all functions available, check the reference.

Trigonometric functions

`sin`(x, /[, out, where, casting, order, ...])	Trigonometric sine, element-wise.
`cos`(x, /[, out, where, casting, order, ...])	Cosine element-wise.
`tan`(x, /[, out, where, casting, order, ...])	Compute tangent element-wise.
`arcsin`(x, /[, out, where, casting, order, ...])	Inverse sine, element-wise.
`arccos`(x, /[, out, where, casting, order, ...])	Trigonometric inverse cosine, element-wise.
`arctan`(x, /[, out, where, casting, order, ...])	Trigonometric inverse tangent, element-wise.
`hypot`(x1, x2, /[, out, where, casting, ...])	Given the “legs” of a right triangle, return its hypotenuse.
`arctan2`(x1, x2, /[, out, where, casting, ...])	Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.
`degrees`(x, /[, out, where, casting, order, ...])	Convert angles from radians to degrees.
`radians`(x, /[, out, where, casting, order, ...])	Convert angles from degrees to radians.
`unwrap`(p[, discont, axis])	Unwrap by changing deltas between values to 2*pi complement.
`deg2rad`(x, /[, out, where, casting, order, ...])	Convert angles from degrees to radians.
`rad2deg`(x, /[, out, where, casting, order, ...])	Convert angles from radians to degrees.

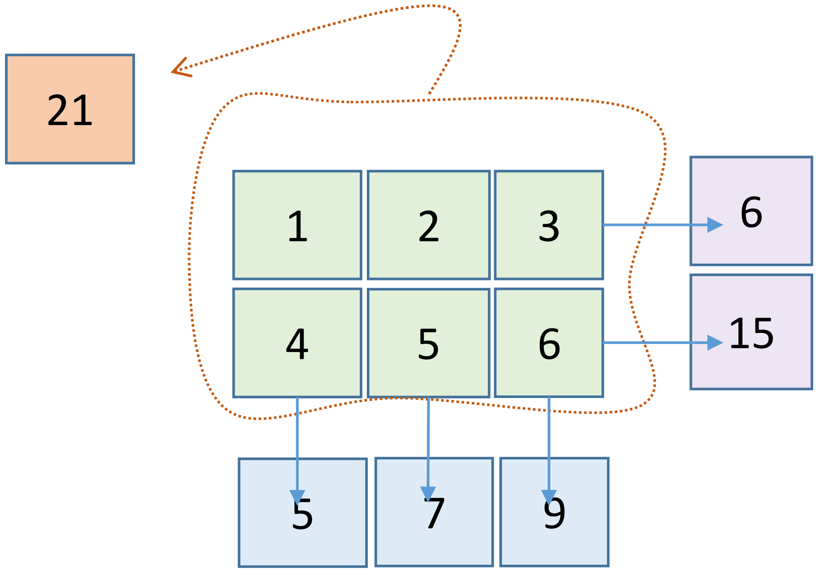
https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.math.html

Sum function: many functions can perform operations in a specific axis

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

z=np.sum(x)
print(z)

m=np.sum(x,axis=0)
print(m)

n=np.sum(x,axis=1)
print(n)
```

OUTPUT

```
21

[5 7 9]

[ 6 15]
```



3D array

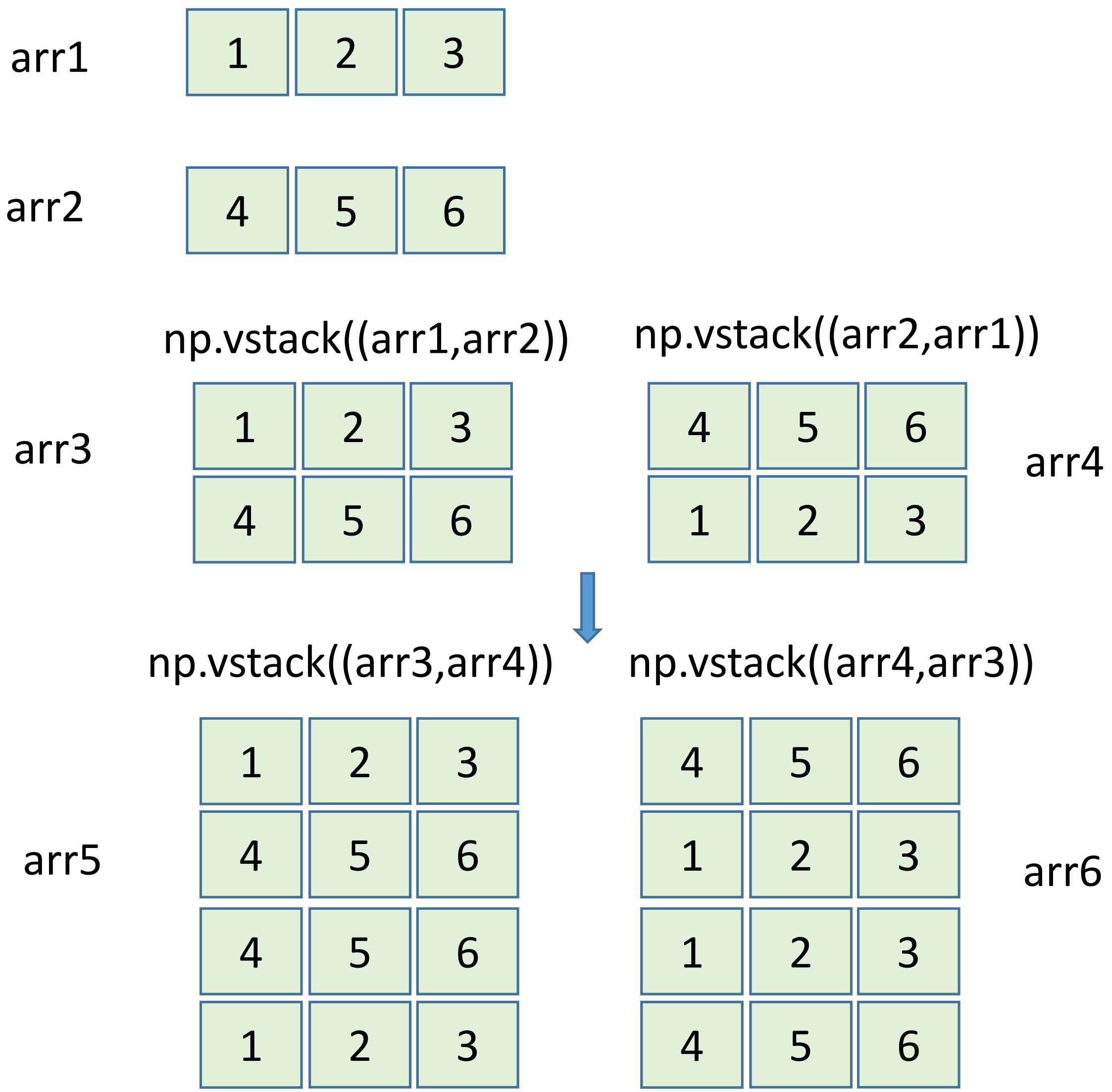
```
xx=np.array([ [[1,2,3], [4,5,6]] , [[7,8,9], [10,11,12]] ] )
p=np.sum(xx,axis=0)
print(p)
```

```
[[ 8 10 12]
 [14 16 18]]
```

Explain yourself, this one

vstack

The vstack() function is used to stack arrays in sequence vertically (row wise)



arr1

1	2	3

arr2

4	5	6

np.vstack((arr1,arr2))

arr3

1	2	3
4	5	6

np.vstack((arr2,arr1))

arr4

4	5	6
1	2	3

np.vstack((arr3,arr4))

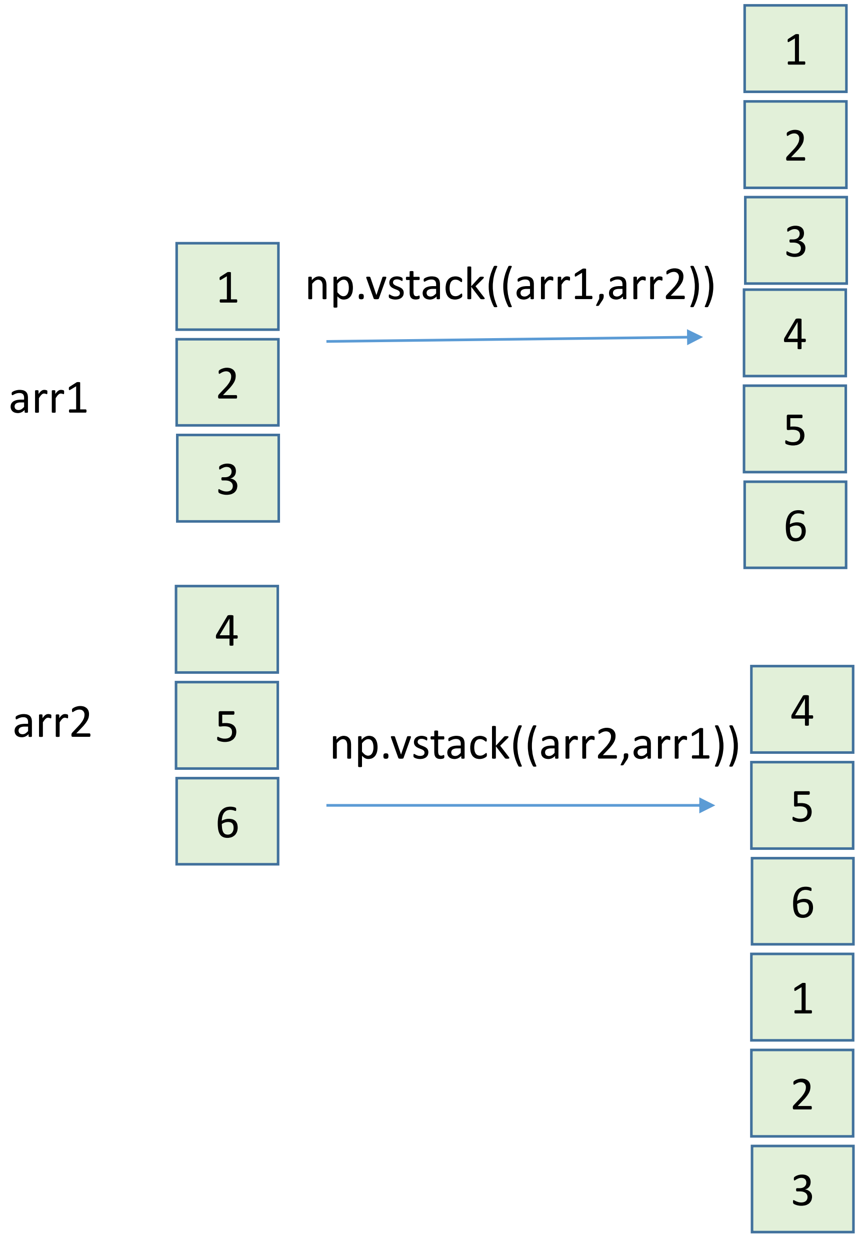
arr5

1	2	3
4	5	6
4	5	6
1	2	3

np.vstack((arr4,arr3))

arr6

4	5	6
1	2	3
1	2	3
4	5	6



arr1

1
2
3

arr2

4
5
6

np.vstack((arr1,arr2))

1
2
3
4
5
6

np.vstack((arr2,arr1))

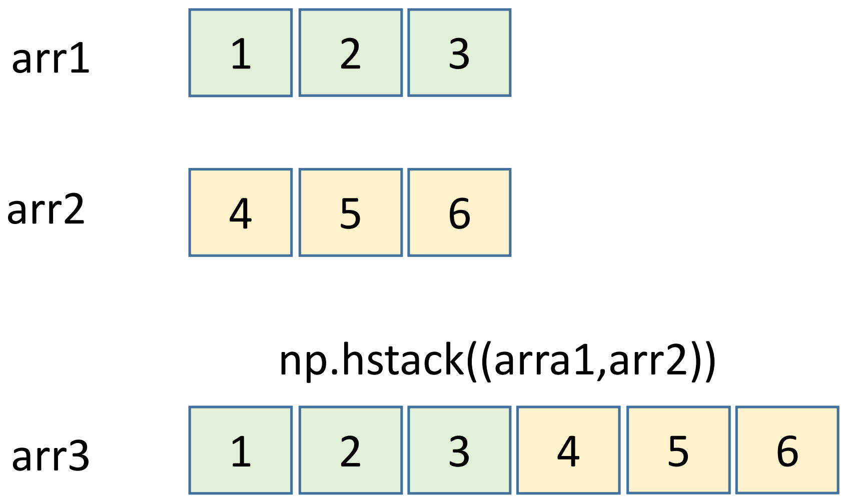
4
5
6
1
2
3

vstack sample code

Code	Output for row vectors	Output for column vectors

import numpy as np
# Row vectors 	
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr3=np.vstack((arr1,arr2))
print(arr3)
arr4=np.vstack((arr2,arr1))
print(arr4)
arr5=np.vstack((arr3,arr4))
print(arr5)
 	
# Column Vectors 	
arr6=arr1.reshape((3,1))
print(arr6)
arr7=arr2.reshape((3,1))
print(arr7)
arr8=np.vstack((arr6,arr7))
print(arr8)
	[[1 2 3]
	 [4 5 6]]
	
	[[4 5 6]
	 [1 2 3]]
	
	[[1 2 3]
	 [4 5 6]
	 [4 5 6]
	 [1 2 3]]
		[[1]
		 [2]
		 [3]]
		
		[[4]
		 [5]
		 [6]]
		
		[[1]
		 [2]
		 [3]
		 [4]
		 [5]
		 [6]]

hstack

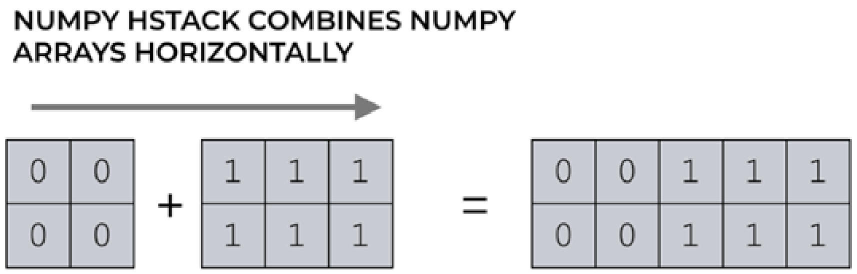


arr1 1 2 3

arr2 4 5 6

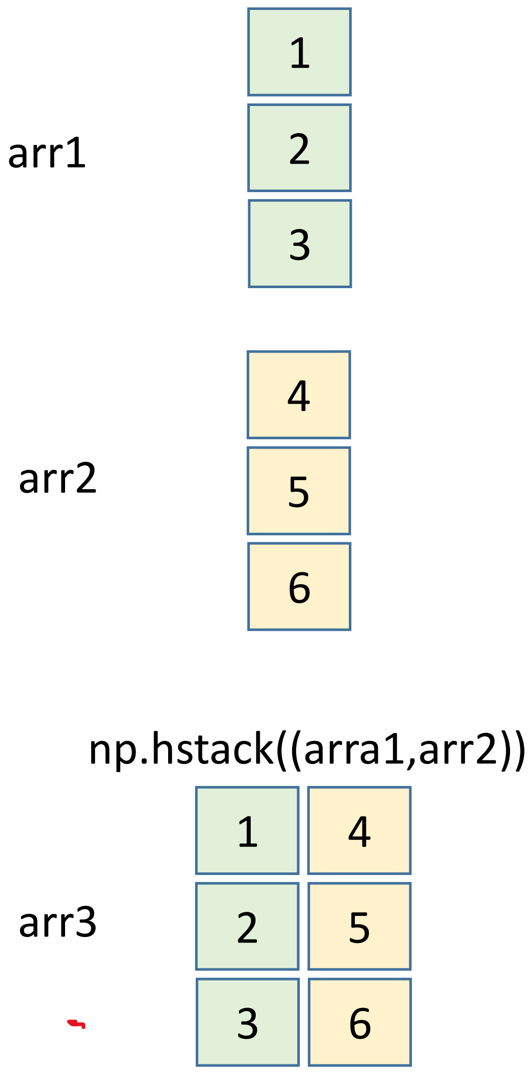
np.hstack((arra1,arr2))

arr3 1 2 3 4 5 6



NUMPY HSTACK COMBINES NUMPY ARRAYS HORIZONTALLY

0 0 + 1 1 1 = 0 0 1 1 1
0 0 + 1 1 1 = 0 0 1 1 1



arr1
1
2
3

arr2
4
5
6

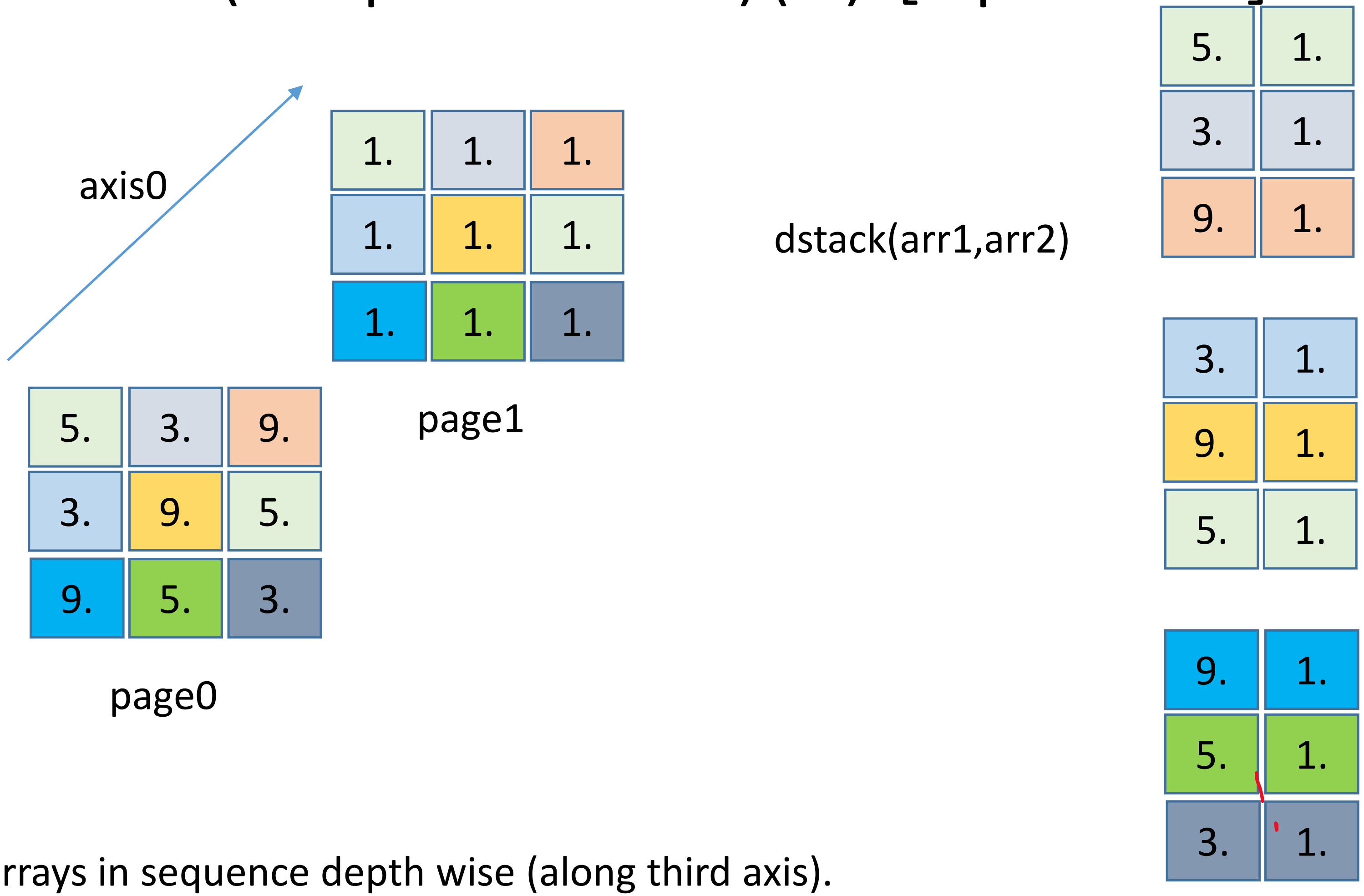
np.hstack((arra1,arr2))

arr3
1 4
2 5
3 6

hstack sample code

Code	Output
import numpy as np	
# Start with row vectors	
arr1 = np.array([1, 2, 3])	
arr2 = np.array([4, 5, 6])	
arr3=np.hstack((arr1,arr2))	
print(arr3)	[1 2 3 4 5 6]
# Start with column vectors	
arr1 = np.array([[1],[2],[3]])	
arr2 = np.array([[4],[5],[6]])	[[1 4]
arr3=np.hstack((arr1,arr2))	[2 5]
print(arr3)	[3 6]]

dstack('depth stack')(1) [optional]



page1

1.	1.	1.
1.	1.	1.
1.	1.	1.

page0

5.	3.	9.
3.	9.	5.
9.	5.	3.

dstack(arr1,arr2)

5.	1.
3.	1.
9.	1.

3.	1.
9.	1.
5.	1.

9.	1.
5.	1.
3.	1.

Stack arrays in sequence depth wise (along third axis).
This is equivalent to concatenation along the third axis after 2-D arrays of shape *(M,N)* have been reshaped to *(M,N,1)*. Or if you start with 1-D arrays of shape *(N,)* have been reshaped to *(1,N,1)*.

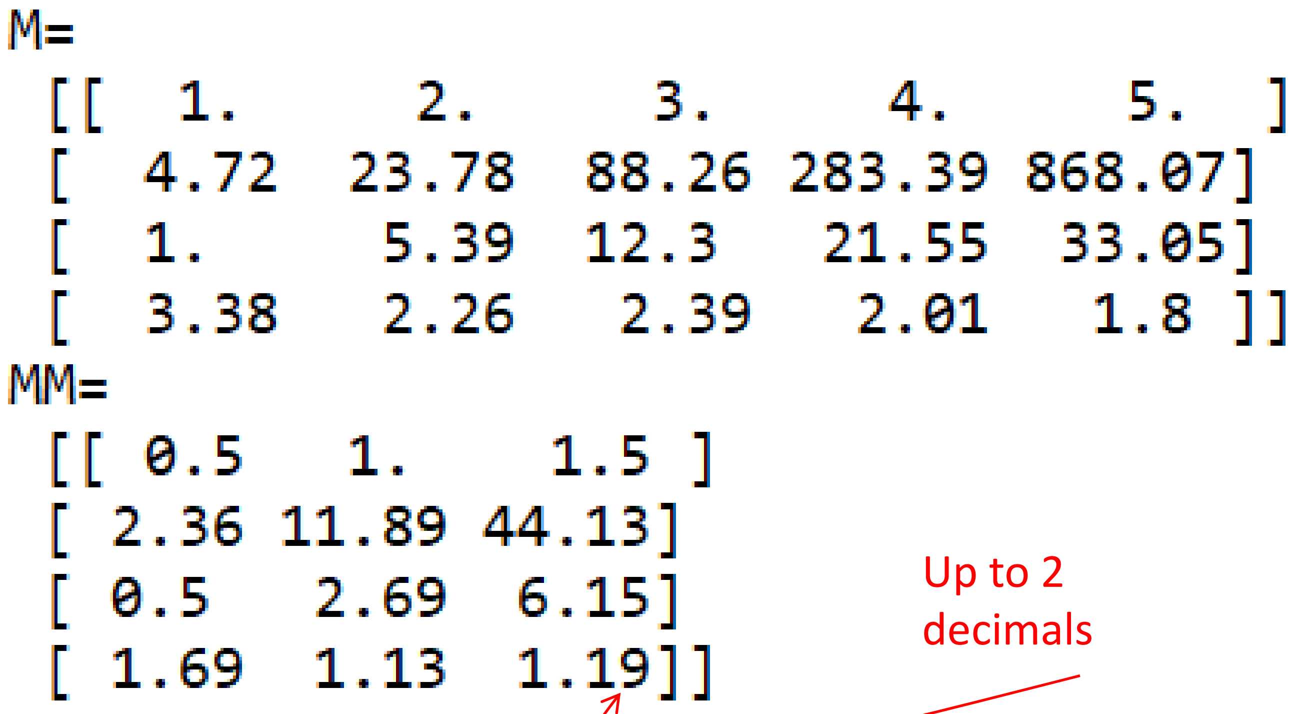
dstack sample code(2) [optional]

Code	Output

import numpy as np
 	
arr0 = np.array([[5,3,9],[3,9,5],[9,5,3]])
arr1 = np.ones((3,3))
 	
arr2= np.dstack((arr0,arr1))
print(arr2)
	[[[5. 1.]
	  [3. 1.]
	  [9. 1.]]
	
	 [[3. 1.]
	  [9. 1.]
	  [5. 1.]]
	
	 [[9. 1.]
	  [5. 1.]
	  [3. 1.]]]

Array Formatted Output: set_printoptions

```
import numpy as np
x=np.array([1,2,3,4,5])
f1=x**3+x*np.exp(x)+1
f2=x**2+x*np.log(x)
f3=np.cos(np.sin(x))+np.exp(1/x)

M=np.vstack((x,f1,f2,f3))
# to set the number of decimals max
np.set_printoptions(precision=2)

print("M= \n",M)
MM=M[:,:3]*0.5
print("MM= \n",MM)
```

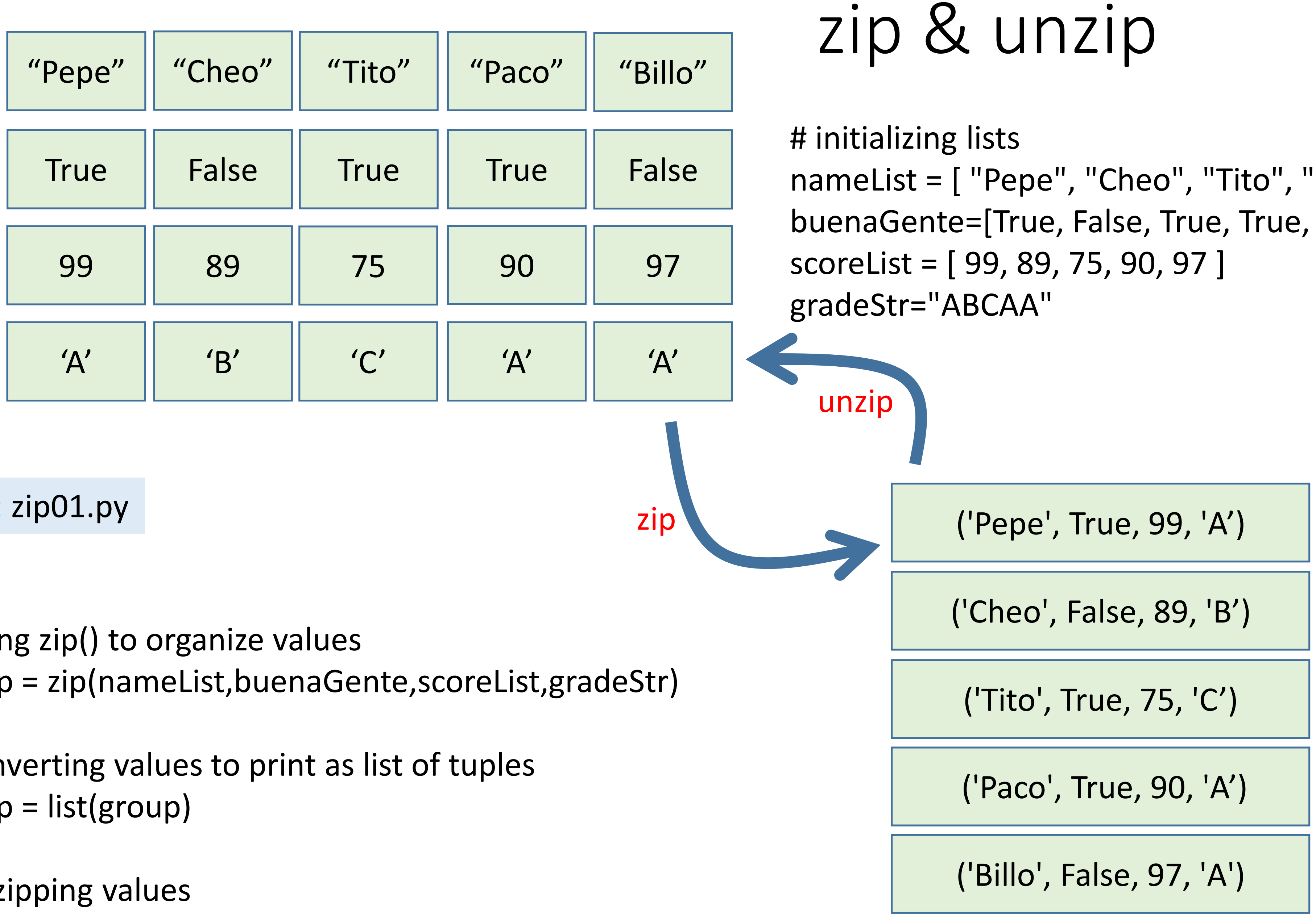


```
M=
 [[   1.      2.      3.      4.      5.  ]
 [   4.72   23.78   88.26  283.39  868.07]
 [   1.      5.39   12.3    21.55   33.05]
 [   3.38    2.26    2.39    2.01    1.8 ]]
MM=
 [[  0.5    1.     1.5 ]
 [  2.36  11.89  44.13]
 [  0.5    2.69   6.15]
 [  1.69   1.13   1.19]]
```

Up to 2 decimals

https://docs.scipy.org/doc/numpy/reference/generated/numpy.set_printoptions.html

zip & unzip



“Pepe”	“Cheo”	“Tito”	“Paco”	“Billo”
True	False	True	True	False
99	89	75	90	97
‘A’	‘B’	‘C’	‘A’	‘A’

('Pepe', True, 99, 'A’)

('Cheo', False, 89, 'B’)

('Tito', True, 75, 'C’)

('Paco', True, 90, 'A’)

('Billo', False, 97, 'A')

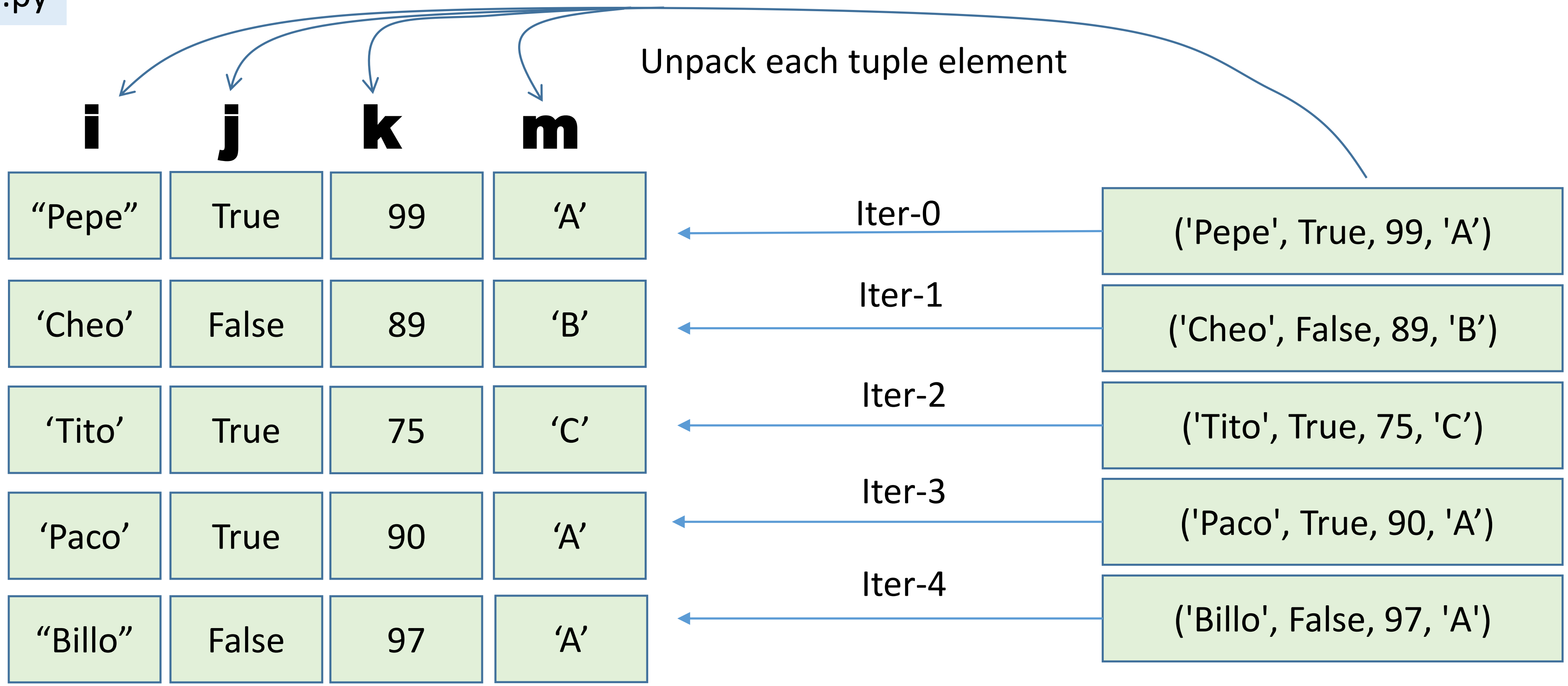
```
# initializing lists
nameList = [ "Pepe", "Cheo", "Tito", "Paco","Billo" ]
buenaGente=[True, False, True, True, False]
scoreList = [ 99, 89, 75, 90, 97 ]
gradeStr="ABCAA"
```

File: zip01.py

```
# using zip() to organize values
group = zip(nameList,buenaGente,scoreList,gradeStr)

# converting values to print as list of tuples
group = list(group)

# unzipping values
nameList, buenaGente, scoreList, gradeStr = zip(*group)
```

```
[('Pepe', True, 99, 'A’),
('Cheo', False, 89, 'B’),
('Tito', True, 75, 'C’),
('Paco', True, 90, 'A’),
('Billo', False, 97, 'A')]
```

File: zip01.py



Unpack each tuple element

i	j	k	m
“Pepe”	True	99	‘A’
‘Cheo’	False	89	‘B’
‘Tito’	True	75	‘C’
‘Paco’	True	90	‘A’
“Billo”	False	97	‘A’

('Pepe', True, 99, 'A’)

('Cheo', False, 89, 'B’)

('Tito', True, 75, 'C’)

('Paco', True, 90, 'A’)

('Billo', False, 97, 'A')

```
for i, j, k, m in zip(nameList, buenaGente, scoreList, gradeStr):
```

# For loop can iterate **simultaneously** over several items in sequences

zip & unzip

What’s n?

Iterate over multiple sequences: zip & unzip

For loop can iterate **simultaneously** over several items in sequences

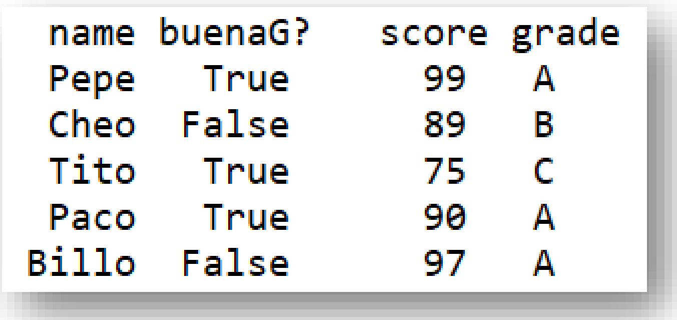
('Pepe', True, 99, 'A’)

('Cheo', False, 89, 'B’)

('Tito', True, 75, 'C’)

('Paco', True, 90, 'A’)

('Billo', False, 97, 'A')



```
   name buenaG?   score grade
   Pepe   True      99   A
   Cheo  False      89   B
   Tito   True      75   C
   Paco   True      90   A
  Billo  False      97   A
```

File: zip01.py

```
import numpy as np

# initializing lists
nameList = [ "Pepe", "Cheo", "Tito", "Paco","Billo" ]
buenaGente=[True,False,True,True,False]
scoreList = [ 99, 89, 75, 90, 97 ]
gradeStr="ABCAA"

# iterating over multiple sequences with one for loop:
print('%7s %6s %7s %3s'%('name','buenaG?','score','grade'))
for i, j, k, m in zip(nameList, buenaGente, scoreList, gradeStr):
    print('%7s %6s %7.0f %3c' %(i, j, k, m))
# if sequences of different length, iterates until the shortest
```

Zip() function(1) (optional)

Organize elements from different sequences into tuples:

```
# initializing lists   [File: zip01.py]
nameList = [ "Pepe", "Cheo", "Tito", "Paco","Billo" ]
rollList= [ 4, 2, 5, 3, 1 ]
scoreList = [ 99, 89, 75, 90, 97 ]
gradeStr="ABCAA"  # or gradeList=['A,'B','C','A','A']

# using zip() to organize values; group is a tuple
group = zip(rollList, nameList, scoreList, gradeStr)

# converting values to print as list of tuples
group = list(group)

# printing resultant values
print("The zipped result is list of tuples :")
print(group)
```

# each tuple element has 4 items

```
The zipped result is a list of tuples :
[ (4, 'Pepe', 99, 'A'), (2, 'Cheo', 89, 'B’),
(5, 'Tito', 75, 'C'), (3, 'Paco', 90, 'A'),
(1, 'Billo', 97, 'A') ]
```

Unzipping (2) (optional)

Unpack elements of tuples:

```
# unzipping values
rollList, nameList, scoreList, gradeStr = zip(*group)
print ("\nThe unzipped result: \n",end=" ")

# printing initial lists
print ("The name list is : ",end=" ")
print (nameList)

print ("The roll list is : ",end=" ")
print (rollList)

print ("The score list is : ",end=" ")
print (scoreList)

print ("The grade string is : ",end=" ")
print (gradeStr)
```

```
The unzipped results:

The name list is : ('Pepe', 'Cheo', 'Tito', 'Paco', 'Billo’)

The roll list is : (4, 2, 5, 3, 1)

The score list is : (99, 89, 75, 90, 97)

The grade string is : ('A', 'B', 'C', 'A', 'A')
```

adiós!