Cheatography

Numpy Cheat Sheet Cheat Sheet

by Taissir Boukrouba (taissir2002) via cheatography.com/193976/cs/40715/

Creating Arrays

Create simple array	np.array([list])
Array filled with zeros	np.zeros(size)
Array filled with ones	np.ones(size)
Array filled with a number	np.full(size,number)
Array from a range of numbers (last not included)	np.arange(start,stop,step)
Array from a range of numbers (last included)	np.linspace(start,stop,count)
Array filled with random numbers (float)	np.random.random(size)
Array filled with random numbers (int)	np.random.randint(start,stop,size)
Create identity Matrix	np.eye(k)
Create empty Matrix	np.empty(k)

Size can be the following :

1) one-dimensional (Array) : where size = k = (k)

eg : np.zeros(2) , creates an array of 2 elements

2) two-dimensional (Matrix) : where size = (a,b)

eg : np.zeros((2,3)) , creates a matrix of 2 rows and one column

3) n-dimensional : where size = (a,b,c....etc) > 2

eg : np.zeros((2,3,5)) , creates a 3 dimensional array

Reading Array From Text File

Inspecting the file (without reading it)	!head file_source	!head data/array_float.csv
Loading File Into Array	np.loadtxt(source, delimiter, dtype)	B = np.loadtxt("data/array_int.csv", delimiter = ",", dtype=int)
Loading Array Into File	np.savetxt(source, array, fmt ,delimiter)	np.savetxt("data/my_array.csv", B, fmt="%3i", delimiter=",")

Make sure that you are typing the right datatype of the text file or you will receive an error

The **fmt** keyword is used to specify the spacing in our text file where :

- `fmt = " %3i "` means 3 spaces between row elements (numbers are integers)
- `fmt = " %5.1 i"` means 5 spaces between row elements and only one number after floating point

Numpy Array Attributes

Returns number of dimensions	array.ndim
Returns size of each dimension	array.shape
Returns number of elements	array.size
Returns type of elements	array.dtype
Returns size of each element in bytes	array.itemsize
Returns size of all elements in bytes	array.nbytes

Array Indexing

Accessing array element	array_name[index]	x1[5]
Modify array element	array_name[index] = new_value	x1[5] = 69

Array Indexing (cont)

Accessing matrix element	matrix_name[row,column]	x2[2, -1]
Modify matrix element	matrix_name[row,column] = new_value	x2[2, -1] = -5
Accessing sub-arrays	array_name[start,stop,step=1]	x[1:5] x[1:5:3]
Accessing multi-dim arrays	array_name[row_range,column_range]	x2[::-1, ::-1]
Creating subarrays as no-copy views	sub_array = original_array[slice]	x2_sub = x2[:2, :2]
Creating subarrays as copy views	sub_array = original_array[slice].copy()	x2_sub_copy = x2[:2, :2].copy()

We can access multiple matrix rows and columns when both row and columns are lists :

x2[[1,2],[1,1]] -> this means access # Second and Third row ([1,2]) , Second and Second column

Reshaping of Arrays

Reshape array	array.reshape((rows,columns))	x2 = x1.reshape((3, 3))
Reshape into row vector	array.reshape((1,columns))	x2 = x1.reshape((1, 3))
Reshape into row vector using np.newaxis	array[np.newaxis, :]	x2 = x1[np.newaxis, :]
Reshape into column vector	array.reshape((rows,1))	x2 = x1.reshape((3, 1))
Reshape into column vector using np.newaxis	array[:, np.newaxis]	x2 = x1[:, np.newaxis]

It is very important to note that number of rows multiplied by the number of columns must be equal to the number of array elements

Array Concatenation and Splitting

Concatenate Arrays vertically (default)	np.concatenate([x1,x2,x3....])	np.concatenate([x, y])
Concatenate Arrays horizontally	np.concatenate([x1,x2,x3....] , axis=1)	np.concatenate([x, y], axis=1)
Stack Arrays vertically	np.vstack([x1, x2,x3....])	np.vstack([x, y])
Stack Arrays horizontally	np.hstack([x1, x2,x3....])	np.hstack([x, y])
Spliting Arrays	np.split(array, [first_split_point, second_split_point , third_split_point....])	x1, x2, x3 = np.split(x, [3, 5])
Splitting Matrixes vertically	np.vsplit(matrix, [frist_row_index,second_row_index....])	upper, lower = np.vsplit(grid, [2])

Array Concatenation and Splitting (cont)

Splitting Matrixes horizontally	np.hsplit(matrix, [frist_col_index,second_col_index....])	left, right = np.hsplit(grid, [2])

NOTE :

concatenate or stack vertically : must have same number of columns

concatenate or stack horizontally : must have same number of rows

NOTE 2 :

The number of split points when splitting an array depends on the number of arrays generated where :

Number of split points = number of arrays outputted - 1

`x1, x2, x3 = np.spl it(x, [3, 5])` -> split on the third element to x1 , split on the fifth element to x2 (from forth element) , give the rest to x3

Numpy Operations & UFuncs

Addition	Array + number	x = [0 1 2 3] x+5 *output : [5 6 7 8]*
Substraction	Array - number	x = [5 6 7 8] x-5 *output : [0 1 2 3]*
Multiplication	Array * number	x*2 *output : [0 2 4 6]*
Division	Array / number	x/2 *output : [0,0.5,1,1.5]*
Floor Division	Array // number	x//2 *output : [0,0,1,1]*
Exponential	Array ** number	x **2* output : [0 1 4 9]*
Modulus	Array % number	x % 2 *output : [0 1 0 1]*
Dot Multiplication	matrix1 @ matrix2	check the notes
UFunc Addition	np.add(array,number)	np.add(x,5) *output : [5 6 7 8]*
UFunc Substraction	np.subtract(array,number)	np.subtract(x,5) *output : [0 1 2 3]*
UFunc Multiplication	np.multiply(array,number)	np.multiply(x,2) *output : [0 2 4 6]*
UFunc Dot	np.dot(matrix1,matrix2)	check the notes
UFunc Absolute Value	np.abs(x)	x = np.array([-2, -1, 0, 1, 2]) np.abs(x) *output : [2,1,0,1,2]*
UFunc Sinus	np.sin(x)	self-explanatory

Numpy Operations & UFuncs (cont)

UFunc Cosinus	np.cos(x)	self-explanatory
UFunc Tangent	np.tan(x)	self-explanatory
UFunc Arcsinus (inverse)	np.arcsin(x)	self-explanatory
UFunc Arccosinus (inverse)	np.arccos(x)	self-explanatory
UFunc Arctangent (inverse)	np.arctan(x)	self-explanatory

For matrix dot operation, the number of columns in the first matrix (array) must be equal to the number of rows in the second matrix (array)

Aggregates Types

array.ufunc.reduce()	This aggregate help apply the universal function between elements and returns (reduces) the result toone value (last value)	x = np.arange(1, 6) np.add.reduce(x) result : 15
array.ufunc.accumulate()	This aggregate help apply the universal function between elements but returns the result of each element inside a list	x = np.arange(1, 6) np.add.accumulate(x) result : array([1, 3, 6, 10, 15])
array.ufunc.outer()	This aggregate help apply the ufunc operation to all pairs (a, b) with a in A and b in B.	np.multiply.outer([1, 2, 3], [4, 5, 6]) result : array([[4, 5, 6],[8, 10, 12],[12, 15, 18]])

The aggregate for multiplication is : **array.multiply.reduce()**
The aggregate for subtraction is : **array.subtract.reduce()**
The aggregate for division is : **array.divide.reduce()**
The aggregate for addition **array.add.reduce()** is equivalent to **np.sum(array)**
The aggregate for multiplication **array.multiply.reduce()** is equivalent to **np.prod(array)**

Other Aggregate Function

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

Other Aggregate Function (cont)

np.percentile	np.nanpercentile	Compute rank-based statistics of elements
np.any(condition)	N/A	Evaluate whether any elements are true
np.all(condition)	N/A	Evaluate whether all elements are true

Python Broadcasting

Broadcasting allows you to perform operations on arrays with different shapes in a way that makes sense, without the need to explicitly reshape or replicate the arrays

Rules of Broadcasting :

Not any operation is permissible when both of arrays have different shapes where :

Rule 01 - Padded Dimensions :

when two have different dimensions we can always add a padding dimension to the left , meaning pushing the first one to the right (5,) => (1,5) (adding new axis)

Rule 02 - Stretching Dimensions :

if any of the aligned dimensions is 1 , then we can stretch that dimension to match the other one

A.shape = (2,3) B.shape = (1,3) => we can stretch B to make it (2,3)

Example :

A.shape = (3, 1)

B.shape = (3,)

We can padd B => B.shape = (1,3)

This means we can strech both A and B (Both have 1 as aligned dimension) where :

A.shape = (3, 1) => (3,3)

B.shape = (1,3) => (3,3)

if any of these rules doesn't apply then broadcasting is not possible

Numpy Fancy Indexing

Select Multiple Elements Using Fancy Indexing	select_elements = array[list_of_indexes]	select_elements = array1[[1, 2, 5, 7]]
Fancy Indexing for Sorting NumPy Array	sorted_array = array[np.argsort(array)]	np.argsort retruns indexes of array sorted (to make it reverse we just write np.argsort(- array))
Fancy Indexing to Assign New Values to Specific Elements	array[list_of_indices] = new_values	array1[[1, 3, 6]] = [10, 20, 30]
Fancy Indexing on N-d Arrays	new_array = array1[list_of_row_indices, list_of_col_indices]	selected_rows = array1[[0,2], :]

In fancy indexing we can either put a list of indices or numpy array of indices

Numpy Sorting

Items sort 1	np.sort(array)	by default it sorts the values in ascending order

Numpy Sorting (cont)		
Items sort 2	array.sort()	this one changes the content of the original array
Index Sorting	np.argsort(array)	this returns the order of the indexes sorted ascendingly
Item Sorting using Index sorting	array[np.argsort(array)]	this returns the items sorted ascendingly
Matrix Sorting	np.sort(matrix,axis=0/1)	this sorts either each matrix columns (axis=0) or each matrix row (axis=1)
Partitioning (Partial Sorting)	np.partition(array, n)	this takes the smallest n numbers to the left in no particular order (3 numbers if n = 3)
Matrix Partitioning	np.partition(array, n, axis=0/1)	this takes the smallest n numbers (of each row or column) to the left in no particular order
Index Partitioning	np.argpartition(array, n, axis=0/1)	this is like argsort() but for matrixes

Array as a Data Structure		
Empty Structured Arrays	np.zeros(struct_size, dtype={'names':(col1,col2,col3), 'formats':(form1, form1, form1)})	data = np.zeros(4, dtype={'names':('name ', 'age', 'weight'), 'formats':('U10', 'i4', 'f8')}) #output : [('', 0, 0.) ('', 0, 0.) ('', 0, 0.) ('', 0, 0.)]

Adding Items to Structured Arrays	# List should be same size as the array structure
	struc_array[colname] = list
		data['name'] = name4
		data['age'] = age4
		data['weight'] = weight4
		#output:
		[('Alice', 25, 55.) ('Bob', 45, 85.5) ('Cathy', 37, 68.) ('Doug', 19, 61.5)]

Accessing array strucure columns	struct_array[colname]	data['name']
Accessing array strucure rows	struct_array[index]	# Accessing last row data[-1]

Array as a Data Structure (cont)		
Sorting array strucure	np.sort(struct_array, order=[coumns])	np.sort(data, order=['age', 'name'])

Partitioning Structured Arrays	np.partition(struct_array, n, order=[columns])	# getting the smallest 2 weights to the left
		np.partition(data, 2, order=['weight'])
		array([('Alice', 25, 55.), ('Doug', 19, 61.5), ('Cathy', 37, 68.), ('Bob', 45, 85.5)]

Defining a new data type	np.dtype([tuple(col_name,type)])	np.dtype([('name', 'U10'), ('age', 'i4'), ('weight', 'f8')])
Appending new row	np.append(struct_array, numpy_array], dtype=dt))	np.append(data, np.array([('John',64,89)], dtype=dt)) [('Alice', 25, 55.) ('Bob', 45, 85.5) ('Cathy', 37, 68.) ('Doug', 19, 61.5) ('John', 64, 89.)]

The Format For Empty Structured Arrays are as follows :
- **Integer** : np.int32 or 'i4'
- **Float** : np.float32 or 'f8'
- **String** : np.str_ or 'U10'