Cheatography

Python For Data Science - NumPy Library Cheat Sheet

by Padma (padma-it) via cheatography.com/121896/cs/22473/

Numpy 1D Arrays

Code	Explanation & OUTPUT
import numpy as np	np is simply an alias
array_1d = np.array([2, 4, 5, 6, 7, 9])	Creating a 1-D array using a list. np.array() takes in a list or a tuple as argument, and converts into an array
print(array_1d)	[2 4 5 6 7 9]
print(type(array_1d))	<class 'numpy.ndarray'>
np.array([iterator, dtype) np.array([1, 2, 3, 4], dtype='float32')	explicitly set the data type array([1., 2., 3., 4.], dtype=float32)
array_from_list = np.array([2, 5, 6, 7]) array_from_tuple = np.array((4, 5, 8, 9))	Convert lists or tuples to arrays np.array(2, 5, 6, 7) will throw an error
list_1 = [3, 6, 7, 5] list_2 = [4, 5, 1, 7]	
array_1 = np.array(list_1) array_2 = np.array(list_2)	Create two 1-D arrays
array_3 = array_1*array_2	Multiple each element of array 1 with array2 **OUTPUT**:[12, 30, 7, 35]
array_4 = array_1 ** 2	[9,36,49,25] # no loop required
np.array([3.14, 4, 2, 3]) Unlike Python lists, NumPy is constrained to arrays that all contain the **same type**	array([3.14, 4. , 2. , 3.]) If types do not match, NumPy will **upcast if possible** e.g. **int upcasted to float**

NumPy Multi-Dimensional Arrays

Syntax and Concepts	Example Code	Explanation & OUTPUT
In NumPy, **dimensions** are called **axes**	array_2d = np.array([[2, 3, 4], [5, 8, 7]])	Creating a 2-D array using two lists
axis = 0 refers to the **rows** axis = 1 refers to the **columns**	print(array_2d) **Note** arrays dont have commas unlike lists on printing	[[2 3 4] [5 8 7]]
np.ones((row_count,column,count),datatype) Default is float	**Create array of 1s** **np.ones((5, 3))** 2D array of axes(5 x 3)of ones	array([[1., 1., 1.], [1., 1., 1.], [1., 1., 1.], [1., 1., 1.], [1., 1., 1.]])
	np.ones((5, 3),dtype=int)	array([[1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]])

NumPy Multi-Dimensional Arrays (cont)		
np.zeros((row_count,column,count),datatype): Default is float	**Create array of 0s** np.zeros(4, dtype = np.int) np.zeros((4,2,2), dtype = np.int)	array([0, 0, 0, 0]) array([[[0, 0], [0, 0]], [[0, 0], [0, 0]], [[0, 0], [0, 0]], [[0, 0], [0, 0]]])
np.random.random(): Default is float	**Create array of random numbers** np.random.random([3, 4])	array([[9.53309987e-01, 7.61005241e-04, 4.11978739e-01, 4.54277232e-01], [6.87842577e-01, 9.02965509e-01, 5.32139081e-01, 5.41951709e-01], [8.58188784e-01, 1.11375267e-01, 8.11638970e-05, 7.04121020e-01]])
np.arange(start,stop,step,dtype): similar to range If **dtype** is not given, infer the data type from other input arguments	**Create array with increments of a fixed step size** numbers = np.arange(10, 100, 5)	[10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95]
np.linspace(start,stop,number of elements,dtype): Default is **float**	**Create array of fixed length;Sometimes, you know the length of the array, not the step size** np.linspace(10, 100, 19,dtype=int)	array([10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100])
np.full((dimensions),element_to_be_filled) Default is **int**	Creating a 4 x 3 array of 7s using np.full(); default is **int** np.full((4,3), 7)	array([[7, 7, 7], [7, 7, 7], [7, 7, 7], [7, 7, 7]])

NumPy Multi-Dimensional Arrays (cont)		
np.tile(arr,repeat_count) Default is **int**	creates a new array by repeating the given array for the given repeated count arr = ([0, 1, 2]) np.tile(arr, 3) np.tile(arr, (3,2))	array([0, 1, 2, 0, 1, 2, 0, 1, 2]) array([[0, 1, 2, 0, 1, 2], [0, 1, 2, 0, 1, 2], [0, 1, 2, 0, 1, 2]])
np.eye(identity_matrix_element,dtype) Default type is **float**	Create a 3 x 3 identity matrix np.eye(3, dtype = int)	array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
np.random.randint(start,stop,(dimensions)) Only integers	rand_array=np.random.randint(0, 10, (4,4))	array([[9, 3, 6, 0], [1, 5, 7, 5], [4, 2, 6, 4], [5, 3, 4, 6]])
Print the second row	print(rand_array[1,])	[1, 5, 7, 5]
np.empty(3)	Create an uninitialized array of three integers	array([1., 1., 1.]) **Could be anything from memory**
	array_1d = np.arange(10) print(array_1d)	[0 1 2 3 4 5 6 7 8 9]
Iteration of 1D array is similar to lists	for i in array_1d: print(i**2)	0 1 4 9
Iterating on 2-D arrays is done with respect to the first axis (which is row, the second axis is column)	for row in array_2d: print(row)	[2 3 4] [5 8 7]
3D array	array_3d = np.arange(24).reshape(2, 3, 4) print(array_3d)	[[[0 1 2 3] [4 5 6 7] [8 9 10 11]] [[12 13 14 15] [16 17 18 19] [20 21 22 23]]]

NumPy Multi-Dimensional Arrays (cont)

Iterating over 3-D arrays: Done with respect to the first axis	for row in array_3d:
	  print(row)
		[[[0 1 2 3]
		  [4 5 6 7]
		  [8 9 10 11]]
		[[12 13 14 15]
		  [16 17 18 19]
		  [20 21 22 23]]]

NumPy Array Attributes

Syntax and Concepts	Example Code	Explanation & OUTPUT
array.shape Returns Shape of array in the form **(n x m)**	print("Shape: {}".format(rand_array.shape))	Shape: (4, 4)
array.ndim Returns number of dimensions (or axes)	print("Dimensions: {}".format(rand_array.ndim))	Dimensions: 2
array.dtype Returns data type (int, float etc.)	print("dtype: {}".format(rand_array.dtype))	dtype: int32
array.size Returns total number of elements in the array	print("Size: ", rand_array.size)	Size: 16
array.itemsize Returns Memory used by each array element in bytes	print("Item size: {}".format(rand_array.itemsize))	Item size: 4
array.nbytes Returns the total size (in bytes) of the array	print("nbytes:", rand_array.nbytes, "bytes")	nbytes: 64 bytes

NumPy Array Indexing

Syntax and Concepts	Example Code	Explanation & OUTPUT
Access single elements: x[start:stop:step] Print third element	print(array_1d[2])	2
Access single elements: x[start:stop:step] Print last element	print(array_1d[-1])	9
Access single element in a 2D array Prints second row third column	print(array_2d[1, 2])	8
Access single element in a 2D array Prints second row last column	print(array_2d[1, -1])	7

NumPy Array Slicing

Syntax and Concepts	Example Code	Explanation & OUTPUT
Accessing subarrays Return specific elements; **Index has to be a list**	x[start:stop:step] print(array_1d[[2, 5, 6]]) print(array_1d[2, 5, 6]) **Default** start: 0, stop: size of dimension, step = 1	[2 5 6] using [] instead of[[]] Will throw error
	array_1d = np.arange(10) print(array_1d)	[0 1 2 3 4 5 6 7 8 9]
Slice third element onwards	print(array_1d[2:])	[2 3 4 5 6 7 8 9]
Slice first three elements	print(array_1d[:3])	[0 1 2]
Slice third to seventh elements	print(array_1d[2:7])	[2 3 4 5 6]
Subset starting 0 at increment of 2	print(array_1d[0::2])	[0 2 4 6 8]
Slicing a 2D array	print(array_2d[1, :])	[5 8 7]
Slicing a 2D array returns an array	print(type(array_2d[1, :]))	<class 'numpy.ndarray'>
Slicing all rows and the third column	print(array_2d[:, 2])	[4 7]
Slicing all rows and the first three columns	print(array_2d[:, :3])	[[2 3 4] [5 8 7]]
Slicing elements within range with step size	print(array_1d[2:7:2])	[2 4 6]
import numpy as np arr1 = np.array([1,2,3,4]) print(arr1) arr2 = arr1[1:] print(arr2) arr2[1] = 8 print(arr1) print(arr2)	Numpy array on slicing will not return new copy. Numpy array slicing will only return a view or reference to original array(like shallow copy)	[1 2 3 4] [2 3 4] [1 2 8 4] [2 8 4]
list1 = [1,2,3,4] print(list1) list2 = list1[1:] print(list2) list2[1] = 8 print(list1) print(list2)	Lists on slicing will create new copy.	[1, 2, 3, 4] [2, 3, 4] [1, 2, 3, 4] [2, 8, 4]

Reshaping of NumPy Arrays

Syntax and Concepts	Example Code	Explanation & OUTPUT
array.reshape(dimensions) Default is **int** x = np.arange(24) print(x) gives [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23] print(x.reshape(3, 2, 4))	**3D array created from 1D array using reshape()** *The last axis has 4 elements, and is printed from left to right.* The second last has 3, and is printed top to bottom * The other axis has 2, and is printed in the two separated blocks	[[[0 1 2 3] [4 5 6 7]] [[8 9 10 11] [12 13 14 15]] [[16 17 18 19] [20 21 22 23]]]
array[np.newaxis,:] creates a row vector **array[:,np.newaxis]** creates a column vector	print(x[np.newaxis,:]) print(x[:,np.newaxis])	[[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]] [[0] [1] [2] .... [22] [23]]

NumPy Array Concatenation

Syntax and Concepts	Example Code	Explanation & OUTPUT
np.concatenate([array1, 2,...],axis=0) Default is along **x-axis** Along x-axis means adding rows(row-wise)(axis=0) Along y-axis means adding columns(xolumn--wise)(axis=1)	x = np.array([1, 2, 3]) y = np.array([3, 2, 1]) z = [99, 99, 99] print(np.concatenate([x, y, z]))	The dimensions should match on the axis the arrays are being concatenated. Here x dimensions are 1 x 3 y dimensions are 1 x 3 z dimensions are 1 x 3 Since columns are same, they can be concatenated across x-axis [1 2 3 3 2 1 99 99 99]
Concatenate on 2D arrays with same dimensions	grid = np.array([[1, 2, 3], [4, 5, 6]]) np.concatenate([grid, grid]) np.concatenate([grid, grid], axis=1)	array([[1, 2, 3], [4, 5, 6], [1, 2, 3], [4, 5, 6]]) array([[1, 2, 3, 1, 2, 3], [4, 5, 6, 4, 5, 6]])

NumPy Array Concatenation (cont)

Concatenate on arrays with different dimensions	np.vstack([arrays]) np.hstack([arrays]) np.dstack([arrays])	
np.vstack([array1, array2]) array1 and 2 should have same column size	print(np.vstack([x,grid]))	[[1 2 3] [1 2 3] [4 5 6]]
np.hstack([array1, array2]) array1 and 2 should have same row size	print(np.hstack([x,y])) y = np.array([[99],[99]]) np.hstack([grid, y])	[1 2 3 3 2 1] [[1 2 3 99] [4 5 6 99]]
np.dstack([arrays])	same as **np.concatenate([arrays],axis=2)	will stack arrays along the third axis

NumPy Array Splitting

Syntax and Concepts	Example Code	Explanation & OUTPUT
np.split(array, [split indices as list])	x = [1, 2, 3, 99, 99, 3, 2, 1] x1, x2, x3 = np.split(x, [3, 5]) print(x1, x2, x3)	Splitting is opposite of Concatenation. N slice points mentioned will give N+1 arrays **[1 2 3] [99 99] [3 2 1]**
	grid = np.array([1,2,3,4,5,6,7,8,9]).reshape((3,3)) print(grid)	
vsplit only works on arrays of 2 or more dimensions	upper, lower = np.vsplit(grid, [2]) print(upper) print(lower)	[[1 2 3] [4 5 6]] [[7 8 9]]
	left, middle, right = np.hsplit(grid, [1,2]) print(left) print(middle) print(right)	[[1] [4] [7]] [[2] [5] [8]] [[3] [6] [9]]

NumPy Aggregation functions

Syntax and Concepts	Example Code	Explanation & OUTPUT
1D array Aggregate operations	m = np.arange(10)	[0 1 2 3 4 5 6 7 8 9]
	print(m)	45
	print(sum(m))	45
	print(np.sum(m))	0
	print(np.min(m))	9
	print(np.max(m))	
2D array aggregate vs normal functions	p = np.arange(9).reshape(3,3)	[[0 1 2]
	print(p)	[3 4 5]
	print(sum(p))	[6 7 8]]
	print(np.sum(p))	[9 12 15]
	print(p.sum())	36
		36
	print(p.min())	0
	print(p.min(axis=0))	[0 1 2]
	print(p.min(axis=1))	[0 3 6]

NumPy Inbuilt Aggregate Functions

Function Name	NaN-safe Version	Description
Most aggregates have a NaN-safe counterpart that computes the result while ignoring missing values,		
np.sum	np.nansum	Compute sum of elements
np.prod	np.nanprod	Compute product of elements
np.mean	np.nanmean	Compute median 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

Mathematical Operations on NumPy Arrays		
Syntax and Concepts	**Example Code**	**Explanation & OUTPUT**
np.sin(array_name) **np.cos(array_name)**	a = np.arange(1, 5) print(np.sin(a)) print(np.cos(a)) print(np.exp(a)) print(np.log(a))	[0.84147098 0.90929743 0.14112001 -0.7568025] [0.54030231 -0.41614684 -0.9899925 -0.65364362] [2.71828183 7.3890561 20.08553692 54.59815003] [0. 0.69314718 1.09861229 1.38629436]
np.vectorize(custom_function)	a = np.arange(5) f = np.vectorize(lambda x: x+10) print(f(a))	[10 11 12 13 14]
Custom function on 2D array Previous functions can be reused for 2D arrays too	b = np.linspace(1, 100, 10,dtype=int) print(b) print(f(b))	[1 12 23 34 45 56 67 78 89 100] [11 22 33 44 55 66 77 88 99 110]
np.linalg	Applies common linear algebra operations	
np.linalg.inv(array_name)	returns array of inverse of a matrix	
np.linalg.det(array_name)	returns determinant value of the matrix	
np.linalg.eig(array_name)	returns eigenvalues and eigenvectors of the matrix	
np.dot(array1,array2))	returns matrix multiplication	