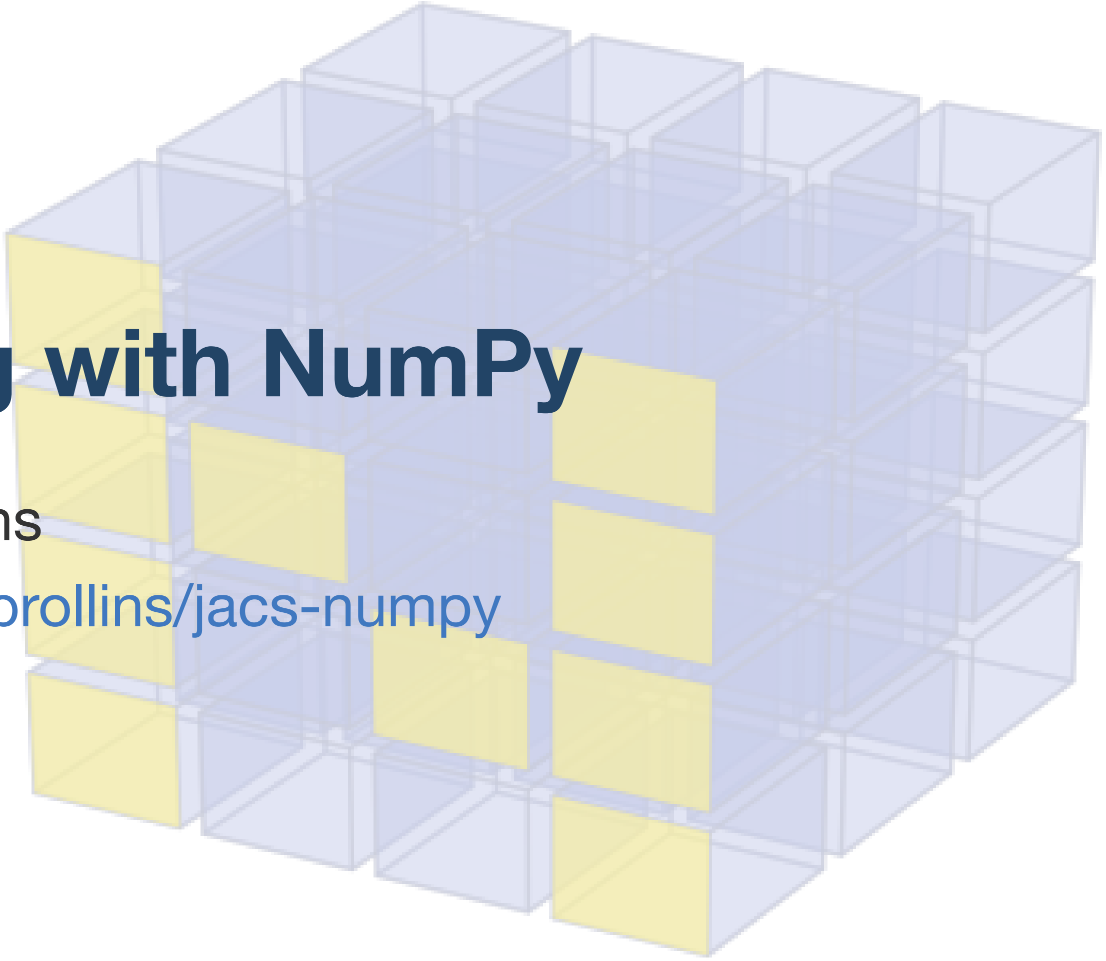


Working with NumPy

Richard Rollins

github.com/rprollins/jacs-numpy

Overview

- Python Performance
- NumPy Arrays
- Universal Functions
- Aggregations
- Broadcasting
- Examples

References

- NumPy Documentation (docs.scipy.org/doc)
- Look Ma, No For-Loops: Array Programming With NumPy - Brad Solomon (realpython.com/numpy-array-programming)
- Losing your Loops Fast Numerical Computing with NumPy - Jake VanderPlas PyCon2015 (youtu.be/EEUXKG97YRw)

Important Convention

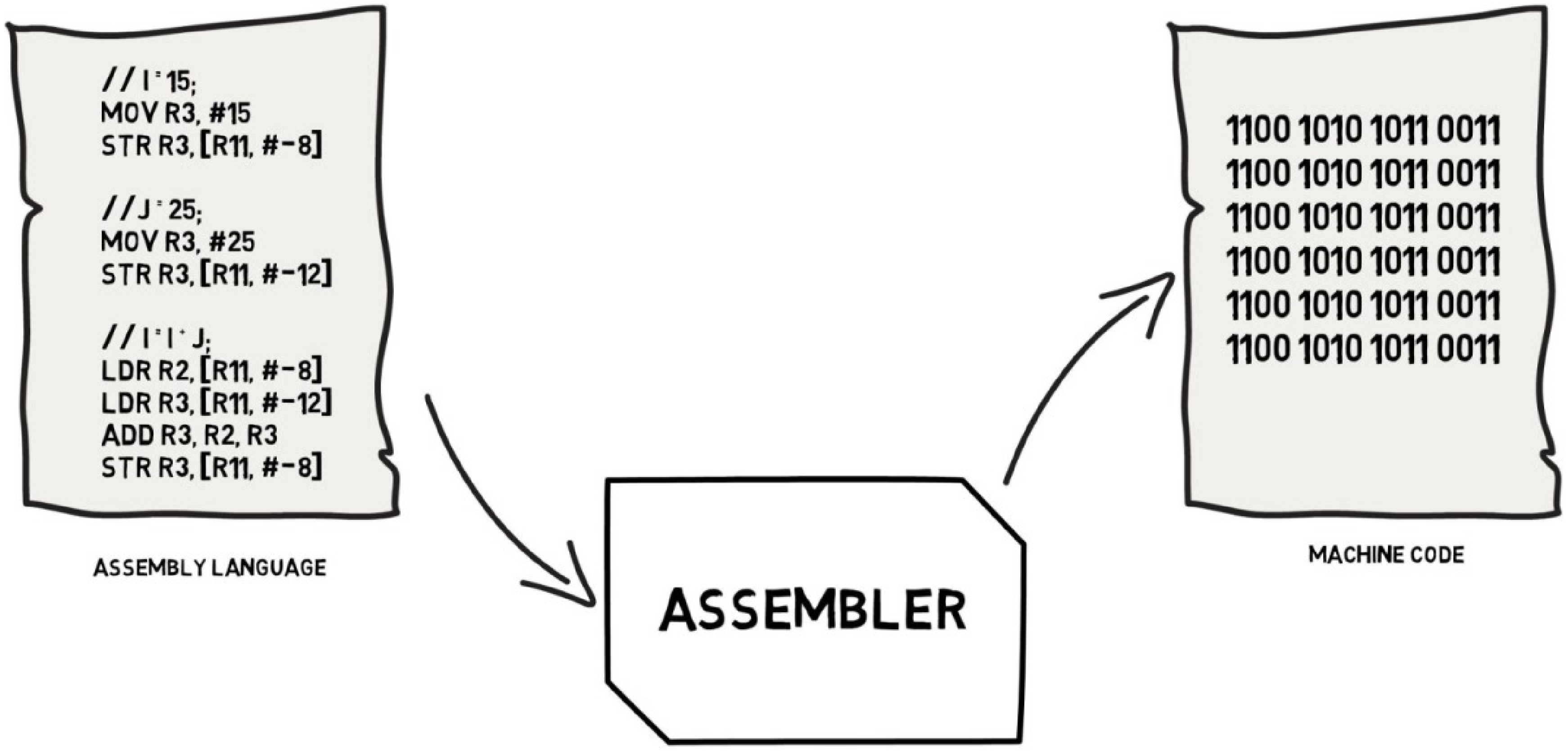
```python
import numpy as np
```

Python Performance

Python is:

- Interpreted - scripts are converted to machine code at runtime
- Dynamically Typed - types are checked for errors at runtime
- High-Level - Fully abstracted from machine code

Assembly & Machine Code



```
// I = 15;
MOV R3, #15
STR R3, [R11, #-8]

// J = 25;
MOV R3, #25
STR R3, [R11, #-12]

// I = I + J;
LDR R2, [R11, #-8]
LDR R3, [R11, #-12]
ADD R3, R2, R3
STR R3, [R11, #-8]
```

```
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
```

Python Performance

"Python is fast for writing, testing and developing code, [but] slow for for repeated execution of low-level tasks."
"What makes python fast (for development) is what makes python slow (for code execution)"
- Jake VanderPlas

Example: Python for loop

```
from random import random
n = 1000000
random_data = [random() for _ in range(n)]
%timeit [x+5 for x in random_data]
```

121 ms ± 274 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

This might sound fast, but:

- My CPU should theoretically do 1 MFlop in roughly 100 µs!
- The equivalent fortran loop took 1.28 ms - 95 times faster!

Why are Python loops slow?

Overheads!

- Type checking
- Pointer dereferencing
- Reference counting

Python lists also cannot utilise vectorised hardware for numerics.

NumPy

`NumPy is best of both worlds: fast development time with fast execution time -> push small repeated executions into statically compiled code and get rid of slow loops`

- Jake VanderPlas

Python Lists vs NumPy Arrays

Python lists are ***general-purpose containers***. They can contain heterogeneous and support efficient insertion, deletion, appending and concatenation. This comes at the cost of overheads and poor performance for repeated low-level operations.

NumPy arrays are ***specialised containers***. They contain homogeneous (numerical) data and have static sizes. However, this allows loops of low-level (numerical) operations over large ammounts of data to be compiled into optimsed C code to be called later without all of the python overheads.

NumPy - Array Creation

From Data:

```
data = np.array([0.1, -1.5, 5.0]) # 1d array of floats
data = np.array([[0, 1], [5, 7]]) # 2d array of integers
```

NumPy - Array Creation

Predefined arrays:

```
data = np.zeros(shape)       # All elements equal zero
data = np.ones(shape)        # All elements equal one
data = np.empty(shape)       # All elements uninitialised
data = np.full(shape,value) # All elements equal 'value'
data = np.identity(size)     # 2d 'identity' array
```

NumPy - Array Creation

Numerical Ranges:

```
data = np.arange([start,]stop[,step,][,dtype])
data = np.linspace(start,stop[,num,endpoint,...])
data = np.logspace(start,stop[,num,endpoint,base,...])
```

NumPy - Properties of ndarrays

Each array object has the following useful member variables:

- size : the total number of elements
- ndim : the total number of dimensions
- shape : the number of elements in each dimension
- dtype : the data type of each element

NumPy - Indexing and Slicing

Individual elements and slices of arrays can be accessed using a similar syntax to lists:

```
data[i]        # Element i of a 1d array
data[i:j:k]    # 1d slice (same syntax as lists)
data[i,j]      # Element i,j of a 2d array
data[i:j:k,l] # Mixed slicing and indexing
```

Note:

- Indexing allows both read and write access
- Indexing multidimensional arrays is comma-separated; this is different from nested lists.

NumPy - Fancy Indexing

We can also use arrays of indices to access and modify complex subsets of an array:

```
>>> data = np.array([1,5,3,9,6])
>>> indices = [0,2,1,4,3]
>>> print(data[indices])
[1 3 5 6 9]
```

Many numpy functions return lists of indices suitable for fancy indexing, including argsort, argwhere, argpartition...

```
>>> data = np.array([1,5,3,9,6])
>>> indices = np.argsort(data)
>>> print(data[indices])
[1 3 5 6 9]
```

NumPy - Masking

We can use arrays of True/False values to access and modify complex subsets of an array:

```
>>> data = np.array([1,5,3,9,6])
>>> mask = [True,False,False,True,False]
>>> print(data[mask])
[1 9]
```

Boolean ufuncs (see next slide) return True/False arrays and can be used to create masks based on logical statements:

```
>>> data = np.array([1,5,3,9,6])
>>> mask = (data < 2) | (data > 8)
>>> print(data[mask])
[1 9]
```

ufuncs

Universal functions perform unary or binary elementwise operations on arrays:

- Arithmetic: + - * / // % **
- Bitwise: & | ~ ^ >> <<
- Inequalities: < > <= >= == !=
- Trigonometric: np.sin, np.cos, np.tan...
- Exponential: np.exp, np.log, np.log10...

Example: Array Addition

```
n = 1000000
data1 = list(range(n))
data2 = list(range(n))
%timeit [x+y for x,y in zip(data1,data2)]
```

149 ms ± 10.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

```
n = 1000000
data1 = np.array(range(n))
data2 = np.array(range(n))
%timeit data1 + data2
```

1.79 ms ± 84 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

A factor of 83x speed-up!

Example: Array Multiplication

```
n = 1000000
data = list(range(n))
%timeit [5*x for x in data]
```

107 ms ± 1.26 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

```
n = 1000000
data = np.array(range(n))
%timeit 5 * data
```

806 µs ± 9.48 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

A factor of 132x speed-up!

Example: Trigonometry

```
from math import sin
n = 1000000
data = list(range(n))
%timeit [sin(x) for x in data]
```

277 ms ± 6.48 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

```
n = 1000000
data = np.array(range(n))
%timeit np.sin(data)
```

12.6 ms ± 2.24 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

A factor of 22x speed-up!

Aggregations

Aggregations perform reductions on arrays and return summary statistics:

- np.min(), np.max()
- np.sum(), np.prod()
- np.mean(), np.std(), np.var()
- np.any(), np.all()
- np.median(), np.percentile()
- np.argmin(), np.argmax()
- np.nanmin(), np.nanmax(), np.nansum() ...

Aggregations can be computed over entire arrays or just a single axis using the 'axis' keyword (see later).

Example: Aggregations

```
from random import random
n = 1000000
data = [random() for _ in range(n)]
%timeit min(data)
```

41 ms ± 1.72 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

```
n = 1000000
data = np.random.random((n))
%timeit data.min()
```

259 µs ± 4.39 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

A factor of 158x speed-up!

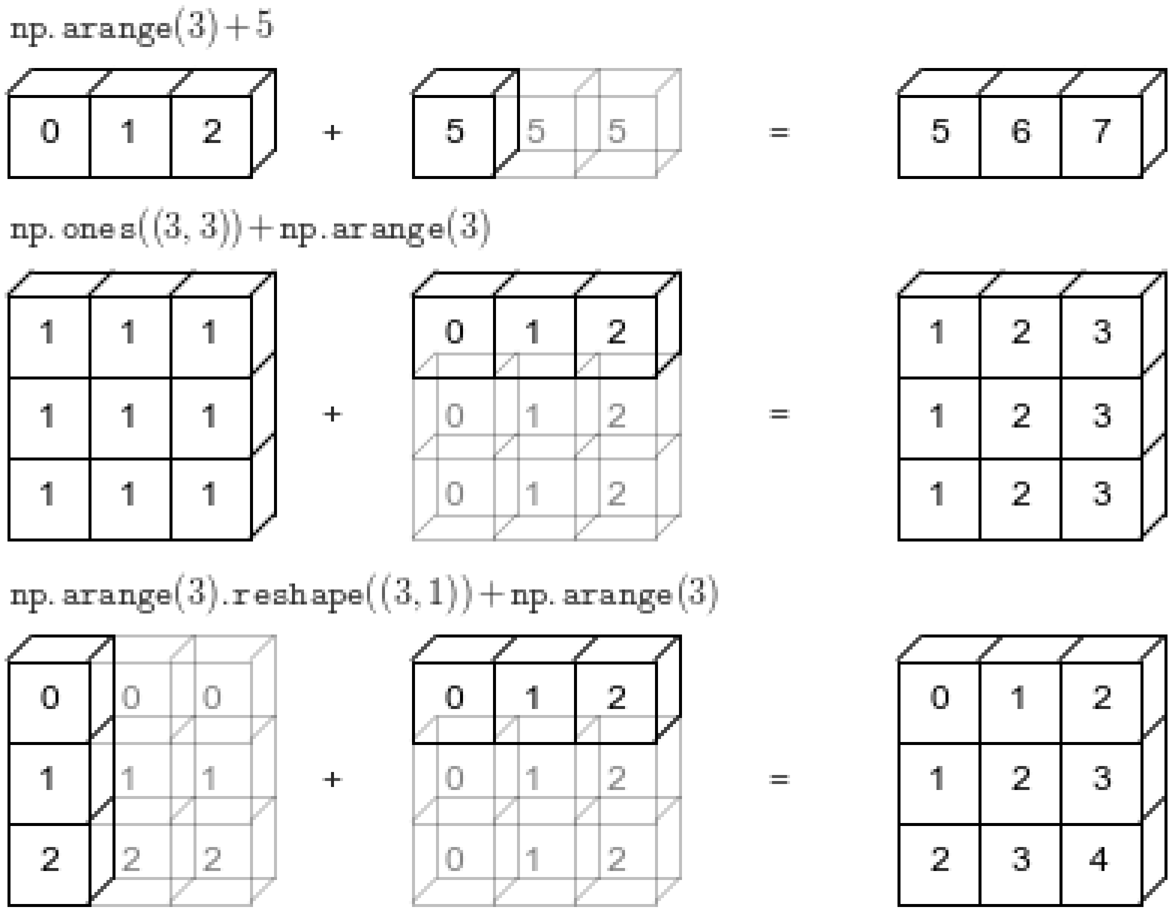
Broadcasting

Broadcasting lets us call binary ufuncs on arrays with different shapes.

To broadcast two arrays, each of their common trailing dimensions must either be the same size or the size of one of them must be one.

Data is then ***implicitly*** repeated along dimensions of size one to match the size of the other array.

Broadcasting



Example: Matrix Multiplication

It is important to stress that numpy.multiply (*) performs elementwise array multiplication ***not*** matrix multiplication. But we can use our knowledge of broadcasting to do our own matrix multiplication:

```
# Example: Multiplying a 2x3 matrix by a 3x5 matrix
matrix1 = np.arange(6).reshape((2,3,1))
matrix2 = np.arange(15).reshape((1,3,5))
result = (matrix1 * matrix2).sum(axis=1)

[[0,1,2],   [[ 0, 1, 2, 3, 4],   [[25,28,31, 34  37],
 [3,4,5]] *  [ 5, 6, 7, 8, 9], =  [70,82,94,106 118]]
             [10,11,12,13,14]]
```

Warning: Do not actually do your own matrix multiplication - use numpy.linalg!

Additional Useful Functions

- np.transpose()
- np.ravel()
- np.reshape()
- np.concatenate()

Example: Nearest Neighbours

Task: Given a set of points in a multidimensional space, what are each points nearest neighbours?

Discuss!

Example: Nearest Neighbours

```python
def nearest_neighbour(p, data):
    deltas = [[x-y for x,y in zip(p,q)] for q in data]
    distances = [sum(x*x for x in d) for d in deltas]
    min_distance = min(filter(lambda x: x>0, distances))
    return distances.index(min_distance)

n = 1000 # Number of points
d = 3    # Number of dimensions
data = [[random() for _ in range(d)] for _ in range(n)]
%timeit [nearest_neighbour(p,data) for p in data]
```

```
2.91 s ± 16.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Example: Nearest Neighbours

```python
def numpy_neighbours(data):
    n = data.shape[0] # Number of points
    d = data.shape[1] # Number of dimensions
    delta = data.reshape((n,1,d)) - data
    distances = np.square(delta).sum(axis=2)
    np.fill_diagonal(distances, np.inf)
    return np.argmin(distances, axis=1)

n = 1000 # Number of points
d = 3    # Number of dimensions
data = np.random.random((n,d))
%timeit numpy_neighbours(data)
```

26.6 ms ± 1.18 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

A factor of 109x speed-up!

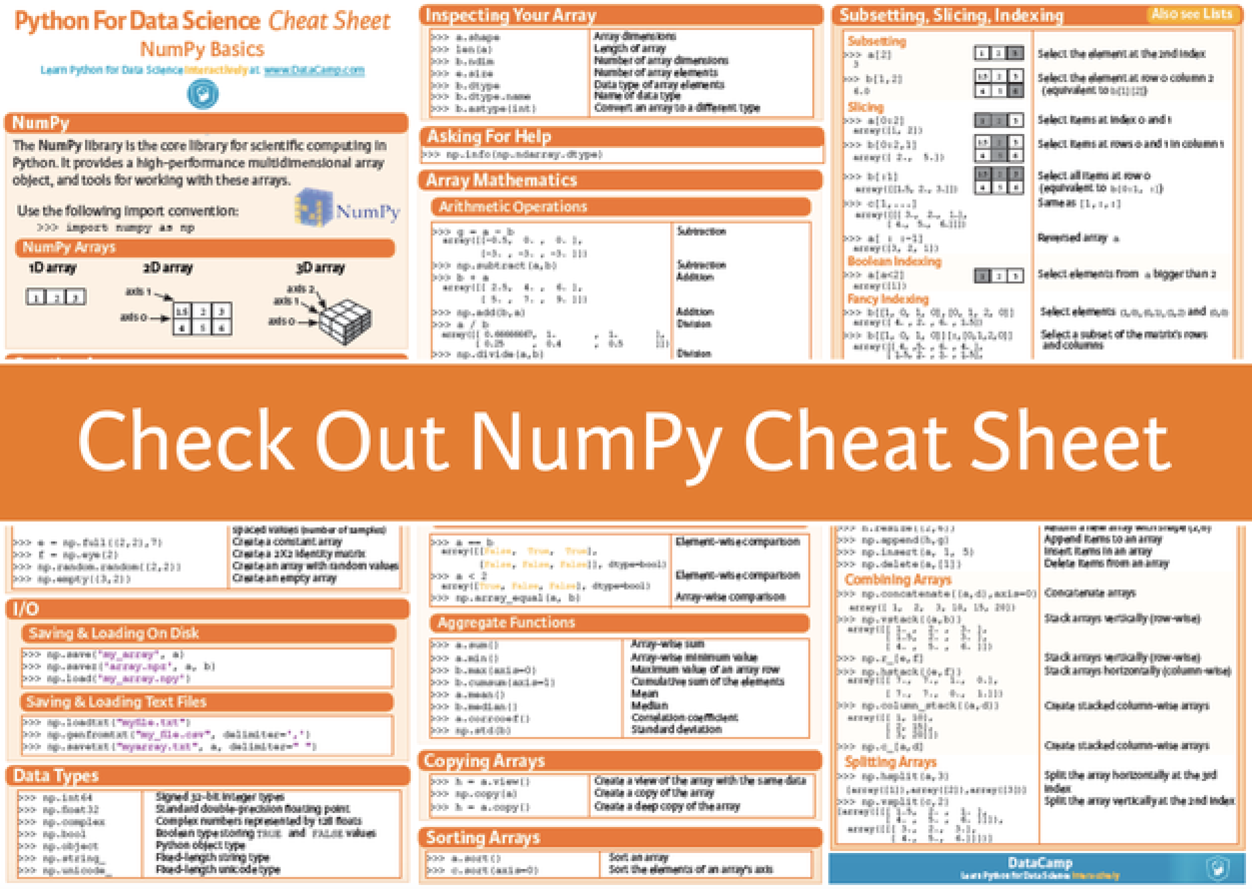
Additional NumPy Modules:

- linalg, fft, random ...

Packages using NumPy:

- SciPy, Matplotlib, Pandas, SKLearn ...

Cheat Sheet



Python For Data Science *Cheat Sheet*

NumPy Basics

Learn Python for Data Science Interactively at www.DataCamp.com

NumPy

The NumPy library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:

```
>>> import numpy as np
```

NumPy Arrays

1D array — 2D array — 3D array

Inspecting Your Array

`>>> a.shape`	Array dimensions
`>>> len(a)`	Length of array
`>>> b.ndim`	Number of array dimensions
`>>> e.size`	Number of array elements
`>>> b.dtype`	Data type of array elements
`>>> b.dtype.name`	Name of data type
`>>> b.astype(int)`	Convert an array to a different type

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

```
>>> g = a - b                          Subtraction
>>> np.subtract(a,b)                   Subtraction
>>> b + a                              Addition
>>> np.add(b,a)                        Addition
>>> a / b                              Division
>>> np.divide(a,b)                     Division
```

Subsetting, Slicing, Indexing — Also see Lists

Subsetting	
`>>> a[2]`	Select the element at the 2nd index
`>>> b[1,2]`	Select the element at row 0 column 2 (equivalent to `b[1][2]`)
Slicing	
`>>> a[0:2]`	Select items at index 0 and 1
`>>> b[0:2,1]`	Select items at rows 0 and 1 in column 1
`>>> b[:1]`	Select all items at row 0 (equivalent to `b[0:1, :]`)
`>>> c[1,...]`	Same as `[1,:,:]`
`>>> a[ : :-1]`	Reversed array `a`
Boolean Indexing	
`>>> a[a<2]`	Select elements from `a` less than 2
Fancy Indexing	
`>>> b[[1, 0, 1, 0],[0, 1, 2, 0]]`	Select elements (1,0),(0,1),(1,2) and (0,0)
`>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]`	Select a subset of the matrix's rows and columns

Check Out NumPy Cheat Sheet

`>>> e = np.full((2,2),7)`	Create a constant array
`>>> f = np.eye(2)`	Create a 2X2 identity matrix
`>>> np.random.random((2,2))`	Create an array with random values
`>>> np.empty((3,2))`	Create an empty array

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

`>>> np.int64`	Signed 64-bit integer types
`>>> np.float32`	Standard double-precision floating point
`>>> np.complex`	Complex numbers represented by 128 floats
`>>> np.bool`	Boolean type storing TRUE and FALSE values
`>>> np.object`	Python object type
`>>> np.string_`	Fixed-length string type
`>>> np.unicode_`	Fixed-length unicode type

`>>> a == b`	Element-wise comparison
`>>> a < 2`	Element-wise comparison
`>>> np.array_equal(a, b)`	Array-wise comparison

Aggregate Functions

`>>> a.sum()`	Array-wise sum
`>>> a.min()`	Array-wise minimum value
`>>> b.max(axis=0)`	Maximum value of an array row
`>>> b.cumsum(axis=1)`	Cumulative sum of the elements
`>>> a.mean()`	Mean
`>>> b.median()`	Median
`>>> a.corrcoef()`	Correlation coefficient
`>>> np.std(b)`	Standard deviation

Copying Arrays

`>>> h = a.view()`	Create a view of the array with the same data
`>>> np.copy(a)`	Create a copy of the array
`>>> h = a.copy()`	Create a deep copy of the array

Sorting Arrays

`>>> a.sort()`	Sort an array
`>>> c.sort(axis=0)`	Sort the elements of an array's axis

`>>> h.resize((2,6))`	Return a new array with shape (2,6)
`>>> np.append(h,g)`	Append items to an array
`>>> np.insert(a, 1, 5)`	Insert items in an array
`>>> np.delete(a,[1])`	Delete items from an array
Combining Arrays	
`>>> np.concatenate((a,d),axis=0)`	Concatenate arrays
`>>> np.vstack((a,b))`	Stack arrays vertically (row-wise)
`>>> np.r_[e,f]`	Stack arrays vertically (row-wise)
`>>> np.hstack((e,f))`	Stack arrays horizontally (column-wise)
`>>> np.column_stack((a,d))`	Create stacked column-wise arrays
`>>> np.c_[a,d]`	Create stacked column-wise arrays
Splitting Arrays	
`>>> np.hsplit(a,3)`	Split the array horizontally at the 3rd index
`>>> np.vsplit(c,2)`	Split the array vertically at the 2nd index

DataCamp

Learn Python for Data Science Interactively

https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf