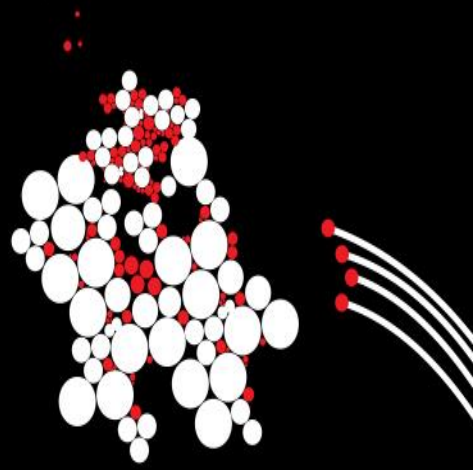
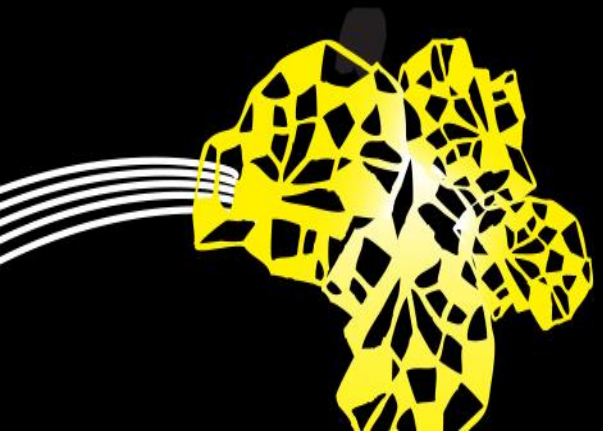
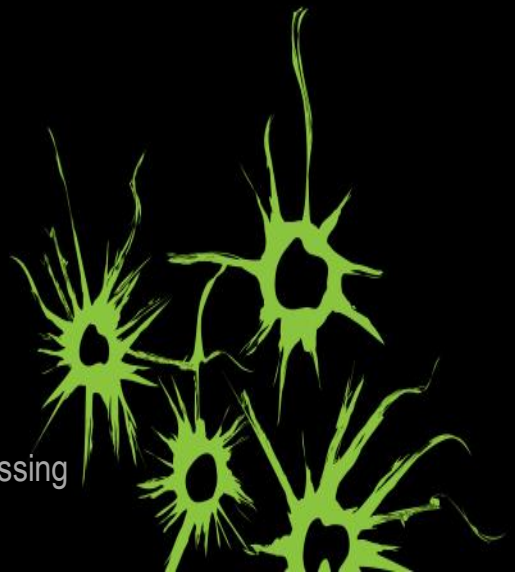
UNIVERSITY OF TWENTE.



NumPy

Array handling with NumPy





Presented by:
Mahdi Farnaghi
Assistant Professor,
Department of Geo-information Processing
ITC, The University of Twente

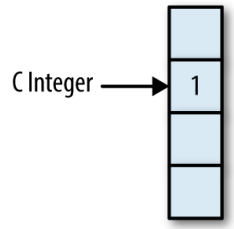
Why NumPy array

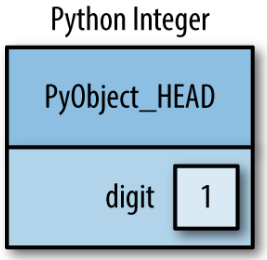
Python and dynamic typing

- Python is well-known for its ease of use
- One of the characteristics: **DYNAMIC TYPING**
- In contrast to strongly typed languages, in Python we do not need to define the type of a variable
- But standard Python implementation is written in C
  - A C integer is a label for a memory position
  - A Python integer is pointer to an object with additional information
  - Additional information allows Python to be dynamic
  - This additional information result in additional cost, in particular for complex data structures

```c
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

```python
# Python code
result = 0
for i in range(100):
    result += i
```



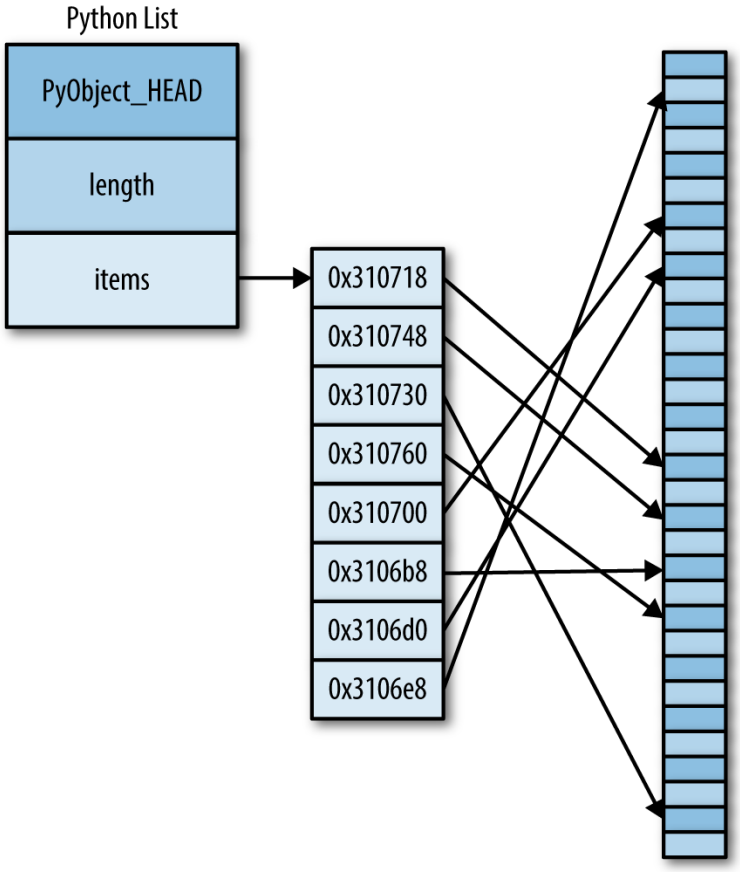


Ref: Python data science handbook, by Jake VanderPlas

Python, dynamic typing, and Python List

- Python List
  - Each item is complete Python object
  - If items in the list are the same, then a lot of redundant data
  - Algorithms need to deal with the additional information of every object in the list
  - **Not Efficient**

- Python answer: Fixed-Type Arrays in Python
- A better solution: NumPy array, which is Fixed-Type and …



Ref: Python data science handbook, by Jake VanderPlas

NumPy

Python library

Typed N-Dimensional Arrays

Fast and versatile, vectorization, indexing, and broadcasting

the de-facto standards of array computing today.

Numerical computing

High-level mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more

Performant

Core: well-optimized C code; Compiled; High speed

Interoperable

Supports a wide range of hardware and computing platforms, and plays well with distributed, GPU, and sparse array libraries

Easy to use

Open source

Array creation

Numpy array creation

We assume throughout these slides that we have
import numpy as np

Array creation

- Conversion from another Python data structure: list or tuple

  np.array([1, 2, 3]) or *np.array((6.05673, 52.56220))*

  - Note: NumPy is contrained to arrays of the same type! NumPy will UPCAST if possible

  array([3.14, 2, 3])

NumPy array creation

- Array generator functions

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

np.arange(6) # half-open interval [0, 6⟩
#  numpy.arange([start, ]stop, [step, ]dtype=None)

np.linspace(0, 1, 12) # closed interval with number of steps
# np.linspace(start, stop, num=50, endpoint=True, retstep=Fal
se, dtype=None)

np.zeros(10, dtype=float)
np.ones((2, 3), dtype=float)
np.full((4,2), 5) # Create a 4×2 array filled with 3.14
```

Other array generators: np.**random**, np.**normal**, np.**eye**, np.**empty**

Array Attributes

```
x1 = np.random.randint(5, size=6)
x2 = np.random.randint(10, size=(3,4))
x3 = np.random.randint(10, size=(2,3,4))
```

```
x1

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

```
x2

array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

```
x3

array([[[8, 1, 5, 9],
        [8, 9, 4, 3],
        [0, 3, 5, 0]],

       [[2, 3, 8, 1],
        [3, 3, 3, 7],
        [0, 1, 9, 9]]])
```

```
print("x1 ndim: ", x1.ndim)
print("x1 shape: ", x1.shape)
print("x1 size: ", x1.size)
print()
print("x2 ndim: ", x2.ndim)
print("x2 shape: ", x2.shape)
print("x2 size: ", x2.size)
print()
print("x3 ndim: ", x3.ndim)
print("x3 shape: ", x3.shape)
print("x3 size: ", x3.size)
```

```
x1 ndim:  1
x1 shape:  (6,)
x1 size:  6

x2 ndim:  2
x2 shape:  (3, 4)
x2 size:  12

x3 ndim:  3
x3 shape:  (2, 3, 4)
x3 size:  24
```

NumPy array attributes

Four important attributes

- ndim
- shape
- size
- dtype

Array Indexing, Slicing, and Reshaping

Array indexing

- Quite similar with Python's standard list indexing

- Takes place between [and]
- Various expression types can be between the brackets
- Is allowed in left-hand and right-hand expression of an assignment

  arrA[ ] = arrB[ ]

Single element indexing

- *arrA[4]* is the 5th element as it is 0-based
- *arrA[-2]* is the one but last element

Reshaping

```
arrA  = np.arange(8)
arrA
array([0,  1,  2,  3,  4,  5,  6,  7])

arrA = arrA.reshape((2,4))                    # just reshape it
arrA
array([[0,  1,  2,  3],
       [4,  5,  6,  7]])

arrA[1,3]                                     # no need to do [1][3]
7                                             # arrA[1][3] still works but is inefficient
arrA[1]                                       # no index for each dimension, so
array([4,  5,  6,  7])                          # lower dimension array returned
```

Observe this … in assignments

```
arrA[1,3] = 8
arrA
array([[0, 1, 2, 3],
       [4, 5, 6, 8]])

arrA[1] = [4,5,6]
Traceback (most recent call last):  File "<stdin>", line 1, in <module>
ValueError: cannot copy sequence with size 3 to array axis with dimension 4

arrA[:,1] = [10,12]                    # works as long as dimension is right
arrA
array([[ 0, 10,  2,  3],
       [ 4, 12,  6,  8]])
```

Observe that arrA and arrA[:,1] share memory on values

Slicing

- In any dimension, a slice is a possibly empty list of indices
- Normal notation is *n* : *k*
  standing for the list of indices from *n* to *k*, *n* inclusive and *k* exclusive.
- If we leave out n, it gets replaced by 0 (read it as "from start")
  If we leave out k, it gets replaced by the highest index value possible (read it as "to end")

- Not often used but allowed is also a stride value (aka step size), with notation *n* : *k* : *s*
  standing for the list of indices going from *n* to *k* with step *s*.
- If we leave out s, we leave out : also, and use 1 as step value.
- Slices and strides allow you to "cut through the data cube."

- Cf. *arange*'s three parameters.

Slicing

```
x = np.arange(10, 30)
```

```
x[::-1]  # all elements, reversed
```

```
array([29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13,
       12, 11, 10])
```

```
x[::2]  # every other element
```

```
array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28])
```

- Multidimensional

```
x = np.arange(0, 24).reshape((4,6))
x
```

```
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]])
```

```
x[:2,:3]
```

```
array([[0, 1, 2],
       [6, 7, 8]])
```

Slicing

```
x[:,1] # Second column
```

```
x[:,1] # Second coloumn
```

```
x[2,:] # Third row
```

```
array([12, 13, 14, 15, 16, 17])
```

A subtle difference between NumPy array and Python list:

NumPy slicing returns ***view*** rather than ***copy***

```
print(x)
x_slice = x[0, 4:]
x_slice[0] = 100
x_slice[1] = 200
print(x)
```

```
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]])
```

```
array([[  0,   1,   2,   3, 100, 200],
       [  6,   7,   8,   9,  10,  11],
       [ 12,  13,  14,  15,  16,  17],
       [ 18,  19,  20,  21,  22,  23]])
```

How to copy? Use copy function! `x[0, 4:].copy()`

Changing the shape of an array

- reshape function
- Add an axis with *np.newaxis* (this adds no values, just reshapes)

- *xx = np.arange(5)*
  xx[:,np.newaxis]
  array([[0],
  [1],
  [2],
  [3],
  [4]])

Concatenation

Joining arrays

- Three important functions
  - np.concatenate()
  - np.vstack()
  - np.hstack()

```
x = np.array([4,5,6])
y = np.array([8,9,10])
np.concatenate([x, y])
```

```
array([ 4,  5,  6,  8,  9, 10])
```

Joining arrays

- Concatenation along other axes

```
z = np.array([[11,22,33],
              [99,88,77]])
```

```
array([[11, 22, 33],
       [99, 88, 77]])
```

```
np.concatenate([z, z]) # concatenate along the first axis
```

```
array([[11, 22, 33],
       [99, 88, 77],
       [11, 22, 33],
       [99, 88, 77]])
```

```
np.concatenate([z, z], axis = 1) # concatenate along the se
cond axis
```

```
array([[11, 22, 33, 11, 22, 33],
       [99, 88, 77, 99, 88, 77]])
```

Universal Functions

Universal Functions

- The importance of NumPy is related to its *vectorized* operations
- Using Python List, operations on arrays need to be done through loop

  e.g., imagine you want to compute the reciprocals of every element in a list, or you want to multiply every element by a constant value.
- Looping over Python List is very slow, due to the dynamic nature of the Python language
  - Type checking
  - Function dispatches
  - …
- Solution
  - NumPy provides a convenient interface over several precompiled array operations
  - This precompiled array operations, known as ***vectorized*** operations, are accessible via NumPy ***ufuncs***
  - Extremely fast, particularly for big arrays
  - No need Loops

UFuncs

- We apply Ufuncs to an array
- The function operates at element level
- With Python List, if we want to calculate the reciprocals of an array:

```
x = [2., 3., 5., 12., 30.]
reciprocals = []
for e in x:
    reciprocals.append(1. / e)
print(reciprocals)
```

- With Ufuncs:

```
x = np.array([2., 3., 5., 12., 30.], dtype=np.float)
reciprocals = 1. / x
print(reciprocals)
```

```
[0.5        0.33333333 0.2        0.08333333 0.03333333]
```

UFuncs

```
x = np.arange(5)
```

We can directly apply Python arithmetic operators as UFuncs to NumPy arrays

```
x = np.arange(5)
print(.3 * x + x**2 + 12)
```

For each of these arithmetic operators there is a function counterpart implemented in numpy

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation
//	Floor division

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

```
[12.  13.3 16.6 21.9 29.2]
```

+	np.add
-	np.subtract
-	np.negative
*	np.multiply
/	np.divide
//	np.floor_divide
**	np.power
%	np.mod

https://jakevdp.github.io/PythonDataScienceHandbook/02.03-computation-on-arrays-ufuncs.html

Other UFuncs

- Trigonometric functions
- Hyperbolic functions
- Exponents and logarithms
- Rounding
- …

sin(x, /[, out, where, casting, order, ...])
cos(x, /[, out, where, casting, order, ...])
tan(x, /[, out, where, casting, order, ...])
arcsin(x, /[, out, where, casting, order, ...])
arccos(x, /[, out, where, casting, order, ...])
arctan(x, /[, out, where, casting, order, ...])

around(a[, decimals, out])
round_(a[, decimals, out])
rint(x, /[, out, where, casting, order, ...])
fix(x[, out])
floor(x, /[, out, where, casting, order, ...])
ceil(x, /[, out, where, casting, order, ...])
trunc(x, /[, out, where, casting, order, ...])

exp(x, /[, out, where, casting, order, ...])
expm1(x, /[, out, where, casting, order, ...])
exp2(x, /[, out, where, casting, order, ...])
log(x, /[, out, where, casting, order, ...])
log10(x, /[, out, where, casting, order, ...])
log2(x, /[, out, where, casting, order, ...])
log1p(x, /[, out, where, casting, order, ...])
logaddexp(x1, x2, /[, out, where, casting, ...])
logaddexp2(x1, x2, /[, out, where, casting, ...])

- See https://numpy.org/doc/stable/reference/routines.math.html for a complete list of available functions

Aggregations

Aggregations

- Simple aggregation functions:

```
np.sum()
np.min()
np.max()
```

- Multidimensional aggregations
  - E.g., how to calculate the maximum value of each row or column in a matrix

```
x = np.arange(1, 16).reshape((5, 3))
print(x)
print("Sum of columns: ", x.sum(axis=0))
print("Sum of rows: ", x.sum(axis=1))
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]]
Sum of columns:  [35 40 45]
Sum of rows:  [ 6 15 24 33 42]
```

List of NumPy aggregation functions

- There are also a NaN-safe version of each of these functions (except for np.all and np.any)
  - E.g., np.nansum()

- NaN: Not a number

Functions	Description
np.mean()	Compute the arithmetic mean along the specified axis.
np.std()	Compute the standard deviation along the specified axis.
np.var()	Compute the variance along the specified axis.
np.sum()	Sum of array elements over a given axis.
np.prod()	Return the product of array elements over a given axis.
np.cumsum()	Return the cumulative sum of the elements along a given axis.
np.cumprod()	Return the cumulative product of elements along a given axis.
np.min(), np.max()	Return the minimum / maximum of an array or minimum along an axis.
np.argmin(), np.argmax()	Returns the indices of the minimum / maximum values along an axis
np.all()	Test whether all array elements along a given axis evaluate to True.
np.any()	Test whether any array element along a given axis evaluates to True.

Broadcasting

Broadcasting

- In NumPy, for arrays of the same size, binary operations perform on an element-by-element basis:

```
x = np.array([10, 20 , 30])
y = np.array([3, 2, 1])
print(x + y)
```

```
[13 22 31]
```

- What happens if we apply a binary operation on two array with different size?

```
a = np.arange(3)
print(a)
print()
b = np.arange(3)[:, np.newaxis]
print(b)
print()
print(a + b)
```

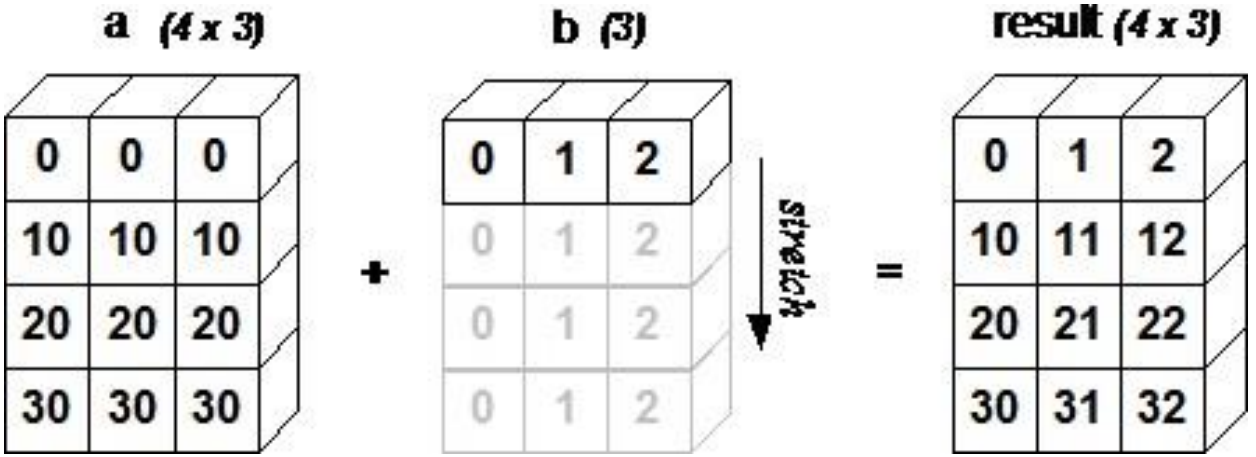
```
[0 1 2]

[[0]
 [1]
 [2]]

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

Cascading

- NumPy tries to stretch and broadcast one of the arrays (or both of them) to match the shape of the other array. Then it applies the binary operation on the output arrays.



https://www.tutorialspoint.com/numpy/numpy_broadcasting.htm

Comparisons and Boolean Mask

Comparison operators on NumPy arrays

- Applying a comparison operator on NumPy arrays result in a Boolean array
  - Boolean array: an array of Boolean values (True/False)
- Example

```
x = np.array([7, 8, 13, 80, 12, 1, 5])
print(x > 14)
```

```
[False False False  True False False False]
```

- Comparison operators: ==, !=, <, <=, >, >=

Boolean masks

Expression *myArr [myBoolArr]* uses another type of indexing with *rather different semantics*.

- The *myBoolArr* must have same shape or be broadcastable to *myArr*.
- Result is a one-dim array with *myArr* values found at positions where *myBoolArr* has value *True*.
- Common example: *myArr [2<myArr & myArr<10]*

Boolean masks can also be used for assignment purposes.

Example

```
x = np.array([7, 8, 13, 80, 12, 1, 5])
b = x >= 13
print(b)
```

```
[False False  True  True False False False]
```

```
y = x[b]
print(y)
```

```
[13 80]
```

Conclusion

Some good references to read

- We only covered the most important aspects of NumPy
- For more information about NumPy functionalities see:
  - https://numpy.org/devdocs/user/quickstart.html
  - https://www.w3schools.com/python/numpy_intro.asp
  - https://www.tutorialspoint.com/numpy/index.htm