NumPy

Big Data
Kristiania University College

Departamento de Automática

Kristiania University College

ISG

Objectives

1. Understand the limitations of plain Python in scientific computation
2. Introduce NumPy
3. Fluent array manipulation with Numpy

Bibliography

Jake VanderPlas. *Python Data Science Handbook*. Chapter 2. O'Reilly. (Link).

Table of Contents

1. Introduction
    - Understanding Data Types in Python
    - NumPy
2. Matrix creation and manipulation
    - Matrix creation
    - NumPy data types
    - NumPy array attributes
    - Accessing single elements
    - Accessing subarrays
    - Reshaping of arrays
    - Concatenation of arrays
    - Splitting of arrays
3. Universal functions
    - Motivation
    - Concept
    - Arithmetic functions
    - Aggregations
    - Broadcasting
4. Comparisons, masks and Boolean logic
    - Motivation
5. Structured arrays

Introduction

Understanding Data Types in Python (I)

Static typing

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

- Data types must be declared
- Data types cannot change
- Error detection in compilation
- Variables names are, basicly, labels

Dynamic typing

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

- Data types are not declared
- Data types can change
- Error detection in run-time
- Variables are complex data structures (even for simple types)

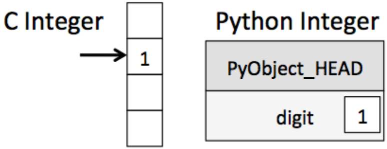
Introduction

Understanding Data Types in Python (II)

Dynamic typing must be implemented somewhere ...

Python 3.4 source code

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

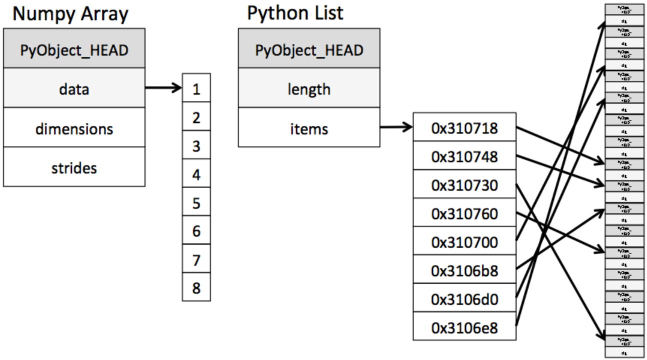


Introduction

Understanding Data Types in Python (III)

A Python list may contain different types

```
In [1]: L3 = [True, "2", 3.0, 4]
   ...: [type(item) for item in L3]
Out[1]: [bool, str, float, int]
```



Introduction

Understanding Data Types in Python (IV)

Standard Python data types are powerful and flexible

- Flexibility has a price: Reduced performance
- Not an big issue in generic programming
- A big issue in scientific programming
- We require efficient data manipulation mechanisms: NumPy

NumPy: Python package for numeric computation

- Efficient array implementation
- Fast mathematical functions
- Random numbers generation
- Static data types: Less flexibility

Most Python modules for AI/ML depend on NumPy, in particular

- Pandas (dataframes), Scikit-learn (ML), Seaborn (data visualization)

Introduction

NumPy

NumPy must be imported in order to be available

- Remember, you can use `np?` or `np.<TAB>`

The main component of NumPy is ndarray

- Python object
- Efficient matrix representation
- Homogeneus elements

Convention

```
^^I^^I^^Iimport numpy as np
^^I^^I^^I
```

```
^^I^^I^^IIn [1]: array = np.
    array([1,2,3])
^^I^^I^^IIn [2]: array
^^I^^I^^IOut[1]: array([1, 2,
     3])
^^I^^I^^IIn [3]: array = np.
    array([[1,2],[3,4]])
^^I^^I^^I
```

Matrix creation and manipulation

Matrix creation

NumPy functions for array creation from lists

- Lists must contain the same type, NumPy will upcast if needed
- `np.array([1, 4, 2, 5, 3])`
- `np.array([1, 2, 3, 4], dtype='float32')`: Explicit data type
- `np.array([3.14, 4, 2, 3])`: Upcast

NumPy functions for array creation from scratch

- `np.zeros(10, dtype=int)`: All zeros
- `np.ones((3, 5), dtype=float)`: All ones
- `np.full((3, 5), 3.14)`: Fill matrix
- `np.arange(0, 20, 2)`: Similar to Python's range()
- `np.linspace(0, 1, 5)`: Evenly spaced numbers
- `np.random.random((3, 3))`: Random numbers
- `np.random.normal(0, 1, (3, 3))`: Random normal numbers
- `np.random.randint(0, 10, (3, 3))`: Random integers
- `np.eye(3)`: Identity matrix
- `np.empty(3)`: Empty matrix

Matrix creation and manipulation

NumPy data types

Python is implemented in C

- Data types in NumPy are based on those in C

Two styles to declare types

- String: `np.zeros(10, dtype='int16')`
- NumPy object: `np.zeros(10, dtype=np.int16)`

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type
`intc`	Identical to C
`intp`	Integer used for indexing
`int8`	Byte
`int16`	Integer
`int32`	Integer
`int64`	Integer
`uint8`	Unsigned integer
`uint16`	Unsigned integer
`uint32`	Unsigned integer
`uint64`	Unsigned integer
`float_`	Shorthand for float64
`float16`	Half precision float
`float32`	Single precision float
`float64`	Double precision float
`complex_`	Shorthand for complex128
`complex64`	Complex number
`complex128`	Complex number

Matrix creation and manipulation

NumPy array attributes

Ndarray objects expose several attributes

- `ndim`: Dimensions
- `shape`: Size of each dimension
- `size`: Number of elements
- `dtype`: Data type
- `itemsize`: Size of each element (in bytes)
- `nbytes`: Size of the array (in bytes)

```
x = np.random.randint(10, size=(3,
    4))
print("x ndim: ", x.ndim)
print("x shape:", x.shape)
print("x size: ", x.size)
print("dtype:", x.dtype)
print("itemsize:", x.itemsize)
print("nbytes:", x.nbytes)
^^I^^I^^I
```

Matrix creation and manipulation

Accessing single elements

Unidimensional array

- `array[index]`

Unidimensional array from the end

- `array[-index]`

Multidimensional array

- `array[row,column]`

```
x = np.array([5, 0, 3, 3, 7, 9])
x[0] # 5
x[4] # 7
x[-1] # 9
x[-2] # 7
x = np.array([[3, 5, 2, 4],
    [7, 6, 8, 8],
    [1, 6, 7, 7]])
x2[2, 0] # 1
x2[2, -1] # 7
^^I^^I^^I
```

Warning

Ndarray has fixed types, values can be truncaded without warning. Big source of problems!

Matrix creation and manipulation

Accessing subarrays

Slice notation can be used with ndarray

- `x[start:stop:step]`

Default values

- Start = 0
- Stop = Size of dimension
- Step = 1

Step may take a negative value

- Reverse order

These operations return a view

- Use `copy()` to get a copy

Unidimensional array

```
x[:5]   # first five elements
x[5:]   # elements after index 5
x[4:7]  # middle sub-array
x[::2]  # every other element
x[1::2] # every other element,
     starting at index 1
x[::-1] # all elements, reversed
^^I^^I^^I
```

Multidimensional a rray

```
x[:2, :3]  # 2 rows, 3 columns
x[:3, ::2] # all rows, every other
     column
x[::-1, ::-1]
^^I^^I^^I
```

Matrix creation and manipulation

Reshaping of arrays

Reshaping arrays is a very common task

- Change data number of dimensions

Important ndarray method: `reshape()`

- Changes the dimensions of an array
- Sizes must match

Conversion of 1-D arrays into column or row matrices

- Using method `reshape()`
- Using the keyword np.newaxis

General reshaping

```
In [1]: x=np.array([1, 2, 3, 4])
In [2]: x.reshape((2,2))
Out[1]:
array([[1, 2],
       [3, 4]])
^^I^^I^^I
```

1-D to row

```
x = np.array([1, 2, 3])
x.reshape((1, 3))
x[np.newaxis, :]
^^I^^I^^I
```

1-D to column

```
x.reshape((3, 1))
x[:, np.newaxis]
^^I^^I^^I
```

Matrix creation and manipulation

Concatenation of arrays

Three methods to join arrays

- `np.concatenate()`
- `np.vstack()`
- `np.hstack()`

np.concatenate()

```
In [1]: x = np.array([1, 2, 3])
In [2]: y = np.array([3, 2, 1])
In [3]: np.concatenate([x, y])
Out[1]: array([1, 2, 3, 3, 2, 1])
^^I^^I^^I
```

np.vstack()

```
In [1]: x = np.array([1, 2, 3])
In [2]: grid = np.array([[9, 8, 7],
   ...:                  [6, 5, 4]])
In [3]: np.vstack([x, grid])
Out[107]:
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
^^I^^I^^I
```

Matrix creation and manipulation

Splitting of arrays

Three methods to split arrays

- `np.split()`
- `np.vsplit()`
- `np.hsplit()`

np.split()

```
^^I^^I^^IIn [1]: x = [1, 2, 3, 99, 99, 3, 2, 1]
^^I^^I^^IIn [2]: x1, x2, x3 = np.split(x, [3, 5])
^^I^^I^^IIn [3]: print(x1, x2, x3)
^^I^^I^^I[1 2 3] [99 99] [3 2 1]
^^I^^I^^I
```

np.vstack()

```
^^I^^I^^IIn [1]: grid = np.arange(16).reshape((4,
    4))
^^I^^I^^IIn [2]: print(grid)
^^I^^I^^I[[ 0  1  2  3]
^^I^^I^^I [ 4  5  6  7]
^^I^^I^^I [ 8  9 10 11]
^^I^^I^^I [12 13 14 15]]
^^I^^I^^IIn [3]: upper, lower = np.vsplit(grid,
    [2])
^^I^^I^^IIn [4]: print(upper)
^^I^^I^^I   [[0 1 2 3]
^^I^^I^^I    [4 5 6 7]]
^^I^^I^^I
```

Universal functions

Motivation

Python may be ridiculously slow

- Run-time type checks and function dispatching
- Evident when an operation is repeated over a collection of data

Performance test

```
def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0 / values[i]
    return output

big_array = np.random.randint(1, 100, size=1000000)
# Stardand CPython
%timeit compute_reciprocals(big_array)
# 3.59 s ± 139 ms per loop
# NumPy
%timeit (1.0 / big_array)
#5.41 ms ± 182 µs per loop
^^I^^I^^I
```

Universal functions (II)

Concept

Vectorized operations: Functions that are aware of NumPy's static typing

- Avoid dynamic type-checking
- Loop related code pushed into the compiled layer
- Hugely improved performance
- Perform an operation with the first element and then it to the rest

In NumPy, vectoriced operations are named universal functions, of ufuncs

- Regular functions
- Arrays as arguments (one or multi-dimensional)
- Operates between arrays of different sizes (broadcasting)

In order to take advantange of NumPy's performance, ufuncs must be used

Universal functions

Arithmetic functions (I)

NumPy makes use of Python's native arithmetic operators

- Used like regular Python operators
- Operators are wrappers for NumPy's functions

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., 1 + 1 = 2)
`-`	`np.subtract`	Subtraction (e.g., 3 - 2 = 1)
`-`	`np.negative`	Unary negation (e.g., -2)
`*`	`np.multiply`	Multiplication (e.g., 2 * 3 = 6)
`/`	`np.divide`	Division (e.g., 3 / 2 = 1.5)
`//`	`np.floor_divide`	Floor division (e.g., 3 // 2 = 1)
`**`	`np.power`	Exponentiation (e.g., 2 ** 3 = 8)
`%`	`np.mod`	Modulus/remainder (e.g., 9 % 4 = 1)

Universal functions

Arithmetic functions (II)

Binary ufuncs

```
x = np.arange(4)
print("x      =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2)  # floor division
np.add(x, 2)               # array plus scalar
^^I^^I^^I
```

Unary ufuncs

```
print("-x      = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)
^^I^^I^^I
```

Universal functions

Basic functions

Absolute value

- `np.absolute(x)` and `np.absolute(x)`

Trigonometric functions

- `np.sin(theta)`, `np.cos(theta)`, `np.tan(theta)`
- `np.arcsin(theta)`, `np.arccos(theta)`, `np.arctan(theta)`

Exponents and logarithms

- `np.exp(x)`, `np.exp2(x)`, `np.power(base, x)`
- `np.log(x)`, `np.log2(x)`, `np.log10(x)`

Advanced mathematical functions

- Checkout module `scipy.special` for exotic mathematical functions

Output as argument

- Avoid temporal variables using `out` argument in ufuncs
- Example: `np.multiply(x, 10, out=y)`

Universal functions

Special functions

Aggregation functions

- Applied to any ufunc
- `reduce(x)`: Repeatedly applies an ufunc to the elements of an array until only a single result remains
- `accumulate(x)`: Like `reduce()`, but it stores intermediate values
- `outer(x)`: Compute the output of all pairs of two different inputs

reduce() example

```
^^I^^I^^IIn [1]: x = np.arange(1, 6)
^^I^^I^^IIn [2]: np.add.reduce(x)
^^I^^I^^IOut[1]: 15
^^I^^I^^I
```

accumulate() example

```
^^I^^I^^IIn [1]: np.add.reduce(x)
^^I^^I^^IOut[1]: 15
^^I^^I^^I
```

Outer() example

```
^^I^^I^^IIn [132]:np.multiply.outer(
    x, x)
^^I^^I^^Iarray([[ 1,  2,  3,  4,
    5],
^^I^^I^^I      [ 2,  4,  6,  8,
    10],
^^I^^I^^I      [ 3,  6,  9, 12,
    15],
^^I^^I^^I      [ 4,  8, 12, 16,
```

Universal functions

Aggregations (I)

Many ufuncs to summarize data

- Basic step in exploratory data analysis
- Argument `axis` determines to which dimension the summary is to be applied

Function	NaN-safe version	Description
`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 standard deviation
`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

Universal functions: Aggregations (II)

(Download dataset)

- Use wget or curl to download the file within iPython

Basic data analysis example

```
^^I import pandas as pd
^^I data = pd.read_csv('president_heights.csv')
^^I heights = np.array(data['height(cm)'])
^^I print(heights)

^^I print("Mean height:       ", heights.mean())
^^I print("Standard deviation:", heights.std())
^^I print("Minimum height:    ", heights.min())
^^I print("Maximum height:    ", heights.max())

^^I print("25th percentile:   ", np.percentile(heights, 25))
^^I print("Median:            ", np.median(heights))
^^I print("75th percentile:   ", np.percentile(heights, 75))
^^I^^I^^I
```

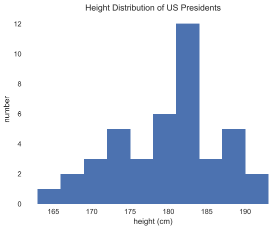
Universal functions

Aggregations (III)

Basic data analysis example (Continuation)

```
^^I^^I^^I%matplotlib inline
^^I^^I^^Iimport matplotlib.pyplot as plt
^^I^^I^^Iimport seaborn; seaborn.set()  # set plot style

^^I^^I^^Iplt.hist(heights)
^^I^^I^^Iplt.title('Height Distribution of US Presidents')
^^I^^I^^Iplt.xlabel('height (cm)')
^^I^^I^^Iplt.ylabel('number');
^^I^^I^^I
```



Universal functions

Broadcasting (I)

Broadcasting is a mechanism to operate over arrays of different sizes

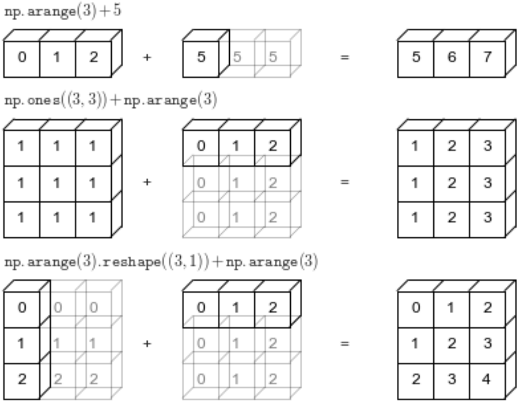
- Used in ufuncs
- Implicit array expansion through three rules

Broadcasting rules

1. Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
2. Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
3. Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Universal functions

Broadcasting (II)



Array expansion does not consume memory!

Universal functions

Broadcasting (III)

Normalization

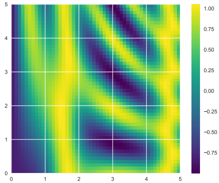
```
^^I^^I^^IX = np.random.random((10, 3))
^^I^^I^^IXmean = X.mean(0)
^^I^^I^^IX_centered = X - Xmean
^^I^^I^^I
```

3D plot

```
^^I^^I^^I%matplotlib inline
^^I^^I^^Iimport matplotlib.pyplot as plt

^^I^^I^^Ix = np.linspace(0, 5, 50)
^^I^^I^^Iy = np.linspace(0, 5, 50)[:, np.newaxis]
^^I^^I^^Iz = np.sin(x)**10+np.cos(10+y*x)*np.cos(x)

^^I^^I^^Iplt.imshow(z, origin='lower',
^^I^^I^^I    extent=[0, 5, 0, 5], cmap='viridis')
^^I^^I^^Iplt.colorbar();
^^I^^I^^I
```



Comparisons, masks and Boolean logic

Motivation (I)

(Download dataset)

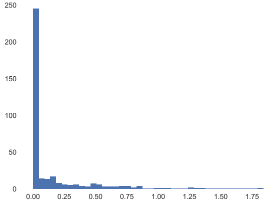
Example

```
^^I^^I^^Iimport numpy as np
^^I^^I^^Iimport pandas as pd

^^I^^I^^I# pandas to extract rainfall inches as a ndarray
^^I^^I^^Irainfall = pd.read_csv('Seattle2014.csv')['PRCP'].values
^^I^^I^^Iinches = rainfall / 254.0  # 1/10mm -> inches
^^I^^I^^Iinches.shape
^^I^^I^^I# Outputs (365,)
^^I^^I^^I
^^I^^I^^I%matplotlib
^^I^^I^^Iimport matplotlib.pyplot as plt
^^I^^I^^Iimport seaborn; seaborn.set()
^^I^^I^^Iplt.hist(inches, 40);
^^I^^I^^I
```

Comparisons, masks and Boolean logic

Motivation (II)



Data filtering is a recurrent task

- How many rainy days were there in the year?
- What is the average precipitation on those rainy days?
- How many days were there with more than half an inch of rain?

Two filtering methods in NumPy

- Boolean arrays masks
- Fancy indexing

Comparisons, masks and Boolean logic

Boolean arrays masks (I)

Syntax examples

```
^^I^^I^^I^^Ix[x<5]
^^I^^I^^I^^Ix[x==3]
^^I^^I^^I^^Ix[(x>3)&(
     x<=5)]
^^I^^I^^I
```

We've seen arithmetic ufuncs ...

- ... but they also support comparison and boolean operations
- Return an array of booleans

Operator	Ufunc
==	`np.equal`
!=	`np.not_equal`
<	`np.less`
<=	`np.less_equal`
>	`np.greater`
>=	`np.greater_equal`

Operator	Ufunc
&	`np.bitwise_and`
|	`np.bitwise_or`
^	`np.bitwise_xor`
~	`np.bitwise_not`

Comparisons, masks and Boolean logic

Boolean arrays masks (II)

Example

```
^^I^^I^^I^^Iprint(x)
^^I^^I^^I^^I[[5, 0, 3, 3]
^^I^^I^^I^^I [7, 9, 3, 5]
^^I^^I^^I^^I [2, 4, 7, 6]]
^^I^^I^^I^^Inp.count_nonzero(x < 6) # Returns 8
^^I^^I^^I^^Inp.sum(x < 6) # Returns 8
^^I^^I^^I^^Inp.sum(x < 6, axis=1) # By row, returns
      array([4,2,2])
^^I^^I^^I^^Inp.any(x > 8) # Returns True
^^I^^I^^I^^Inp.any(x < 0) # Returns False
^^I^^I^^I^^Inp.all(x < 10)# Returns True

^^I^^I^^I^^Inp.sum(~((inches <= 5) | (inches >= 1))
     )
^^I^^I^^I
```

Comparisons, masks and Boolean logic

Fancy indexing

So far we've seen three accessing methods

- Simple indices (`x[1]`)
- Slices (`x[:5]`)
- Boolean masks (`x[x>0]`)

Fancy indexing: Pass arrays on indices instead of scalars

Example

```
^^I^^I^^I^^Ix = rand.randint(100, size=10)
^^I^^I^^I^^I[x[3], x[7], x[2]] # Simple
     indices
^^I^^I^^I^^Iind = [3, 7, 4] # Array of
     indices
^^I^^I^^I^^Ix[ind] # Fancy indexing
^^I^^I^^I^^Ix[[3,5,6]] # Also valid
^^I^^I^^I
```

The shape of the result reflects the shape of the index arrays rather than the shape of the array being indexed

Structured arrays (I)

Some times, we need to group data

- Example: Store name, age and weight of several people
- Different data types for each attribute

Non-structured array

```
^^I^^I^^Iname = ['Alice', 'Bob', 'Cathy', 'Doug']
^^I^^I^^Iage = [25, 45, 37, 19]
^^I^^I^^Iweight = [55.0, 85.5, 68.0, 61.5]
^^I^^I
```

Solution: Structured arrays

Structured arrays

```
^^I^^I# Use a compound data type for structured arrays
^^I^^Idata = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                          'formats':('U10', 'i4', 'f8')})
^^I^^I
```

Structured arrays (II)

Structured array manipulation

```
^^I^^Idata['name'] = name
^^I^^Idata['age'] = age
^^I^^Idata['weight'] = weight

^^I^^I# Get all names
^^I^^Idata['name']
^^I^^I# Get first row of data
^^I^^Idata[0]
^^I^^I# Get the name from the last row
^^I^^Idata[-1]['name']
^^I^^I# Get names where age is under 30
^^I^^Idata[data['age'] < 30]['name']
^^I^^I
```

These kind of structures are day-to-day used

- Pandas is a much better choice