

# Scientific Programming in Python

Inteligencia Artificial en los Sistemas de Control Autónomo  
Máster Universitario en Ingeniería Industrial

Departamento de Automática

## Objectives

1. Introduce some Python tools for scientific programming.
2. Motivate the need of efficient matrix manipulation.
3. Handle matrices and dataframes in Python.
4. Basic data visualization with Python.

## Bibliography

Jake VanderPlas. Python Data Science Handbook. Chapters 1, 2, 3 and 4. O'Reilly. (Link).

# Table of Contents

## 1. Overview

- Data Science
- The data scientist toolkit
- Anaconda
- Python IDEs for Data Science

## 2. Basics

- Magic commands
- Pasting code blocks
- Running external code
- Input and output history
- iPython shell commands
- Automagic

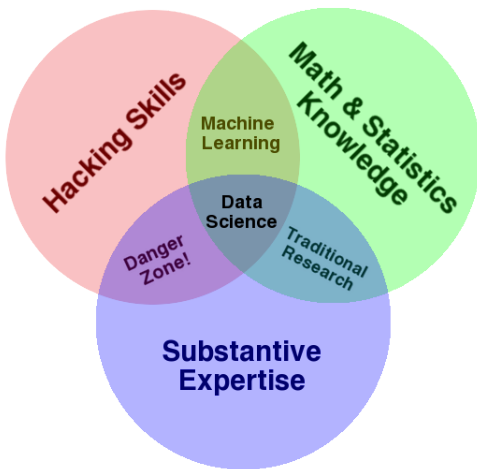
## 3. NumPy

- Understanding Data Types in Python
- Introduction
- Matrix creation
- NumPy data types
- NumPy array attributes
- Accessing single elements
- Accessing subarrays
- Reshaping of arrays
- Concatenation of arrays
- Splitting of arrays
- Universal functions
- Aggregations
- Broadcasting
- Structured arrays

## 4. Pandas

- ASDF

## Data Science



# Data Science

## The data scientist toolkit (I)

## Data science is about manipulating data

- Need of specialized tools
- Two main languages: R and Python

# Python is a general purpose programming language

- Easy integration
- Huge ecosystem of packages and tools

## Need of data-oriented tools

- Features provided by third-party tools

# Data Science

## The data scientist toolkit (II)

Tool	Type	Description
iPython	Software	Advanced Python interpreter
Jupyter	Software	Python notebooks (Python interpreter)
Numpy	Package	Efficient array operations
Pandas	Package	Dataframe support
Matplotlib	Package	Data visualization
Seaborn	Package	Data visualization with dataframes
Scikit-learn	Package	AI/ML package for Python

# Data Science

# Anaconda

All those tools are packaged in Anaconda

- Python distribution for Data Science

## Anaconda provides Spyder

- Python IDE designed for Data Science

## Other tools provided by Anaconda

- Conda: Packages management tool
- TensorFlow: Deep Learning
- Many others



# Data Science

## Python IDEs for Data Science (I)

## iPython

iPython = Interactive  
Python

- Extended functionality
- Enhanced UI
- External editor

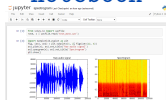
Running iPython:  
\$ ipython

## Jupyter

## Python notebooks

- Web-based IDE
- Documentation
- Integration with GitHub
- Uses iPython

Running Jupyter:  
\$ jupyter  
notebook



## Rodeo

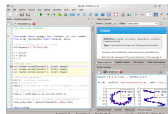
## Python version of RStudio

- Good for R developers
- Not included in Anaconda
- Uses iPython



## Spyder

## Matlab-like IDE





# Data Science

## Python IDEs for Data Science (II)

## Exercises

Write a Python script that shows the multiplication table of the number 5. Write the script using each one of the following environments:

1. iPython + text editor of your choice.
2. Jupiter.
  - Bonus track: Publish the notebook in GitHub.
3. Spyder.
4. Rodeo.

## iPython

## Basics (I)

In regular Python ...

- most objects come with a docstring attribute
- docstring accesible thorough `help()`

iPython provides `'?'`, a shortcut to `help()`

- `len?`, `list?`, `list.append?`
- Try to type just `'?`

Easy access to source code with `??`

- Does not work with most builtin functions!

# iPython

## Basics (II)

Press <tab> to complete almost everything

- Object contents

```
In [21]: a = [1,2,3]
In [22]: a.
```

a.append	a.count	a.insert	a.reverse
a.clear	a.extend	a.pop	a.sort
a.copy	a.index	a.remove	

- Packages

```
In [26]: import num
```

- Wildcards

```
In [29]: *Warning?
```

%%!	BaseException
ArithmeticError	BlockingIOError
AssertionError	BrokenPipeError
AttributeError	BufferError

## iPython

## Basics (III): Keyboard shortcuts

## Navigation

KEYSTROKE	ACTION
Ctrl-a	Move cursor to the beginning of the line
Ctrl-e	Move cursor to the end of the line
Ctrl-b	Move cursor back one character
Ctrl-f	Move cursor forward one character

## History

KEYSTROKE	ACTION
Ctrl-p (↑)	Previous command
Ctrl-n (↓)	Next command
Ctrl-r	Reverse-search

## Text entry

KEYSTROKE	ACTION
Ctrl-d	Delete next character in line
Ctrl-k	Cut text from cursor to end of line
Ctrl-u	Cut text from beginning of line to cursor
Ctrl-y	Yank (paste) previously cut text

## iPython

## iPython magic commands

## Magic commands: iPython extension of Python syntax

- Not valid in regular Python
- Provides handy features
- Widely used in DS and ML

## Two flavours

- % prefix: Line magics - single line
- %% prefix: Cell magics - several lines

Help available

- %magic: Magic commands
- %lsmagic: List of magic commands









## iPython

## Input and output history (II)

Fast access to history: Underscore (  )

- Variable containing the last output
- Example: `print(_)`

## Double and triple underscores

- Example: `print(__)`
- Example: `print(____)`

Trick: Shortcut to access `(_n)`

- Out[n] = \_n, with n=number
- Example: `print(_2)`

## Magic command to show history

- %history

### Supressing command output (;)

- Example:  $4 * 2$ ;

## iPython

## iPython shell commands

iPython provides easy interaction with the shell

- Execution of shell commands from iPython
- Use prefix `!`
- Example: `!ls`, `!pwd`

## Save shell output in Python variables

- Example: `files = !ls`

## Use Python variables in shell

- Example: `!echo {files}`

iPython  
Automagic

## Problems with some shell commands

```
In [23]: !pwd
/repositorios/pythonCourse
In [24]: !cd ..
In [25]: !pwd
/repositorios/pythonCourse
```

Some magic commands here to help

- %cd,%ls,%mkdir,%pwd,  
...

Those magics are regularly used ...

- ... so common that % is no longer required (automagic)
- Working with iPython is almost like working with a Unix-like shell

## Automagic commands

cat, cp, env, ls, man, mkdir, more,  
mb, pwd, rm and rmdir

## NumPy

# 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, basically, 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)

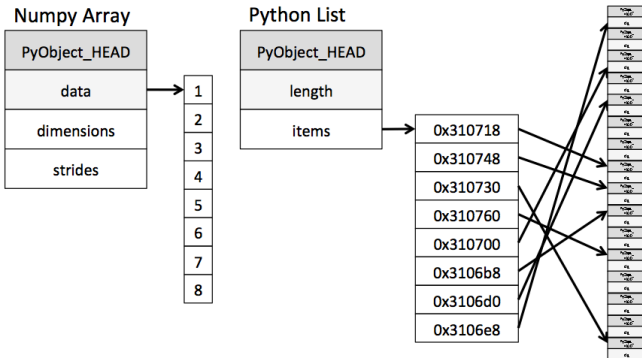


## NumPy

## 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]
```



## NumPy

## 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)

# NumPy

## Introduction

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
- Homogeneous elements

## Convention

```
import numpy as np
```

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



# NumPy

## 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

# NumPy

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



# NumPy

## 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
```

### Warning

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

# NumPy

## 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
```

### Multidimensional array

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

# NumPy

## 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]])
```

### 1-D to row

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

### 1-D to column

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

# NumPy

## 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])
```

`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]])
```

# NumPy

## Splitting of arrays

Three methods to split arrays

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

`np.split()`

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

`np.vstack()`

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



# NumPy

## Universal functions (I)

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)
# Standard 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
```

# NumPy

## Universal functions (II)

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 advantage of NumPy's performance, ufuncs must be used

# NumPy

## Universal functions: Arithmetic functions

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

# NumPy

## Universal functions (III)

### 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
```

### Unary ufuncs

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

# NumPy

## Universal functions: Basic functions

### Absolute value

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

# NumPy

## 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

```
In [1]: x = np.arange(1, 6)
In [2]: np.add.reduce(x)
Out [1]: 15
```

#### accumulate() example

```
In [1]: np.add.reduce(x)
Out [1]: 15
```

#### Outer() example

```
In [132]: np.multiply.outer(x, x)
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

# NumPy

## 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
<code>np.sum</code>	<code>np.nansum</code>	Compute sum of elements
<code>np.prod</code>	<code>np.nanprod</code>	Compute product of elements
<code>np.mean</code>	<code>np.nanmean</code>	Compute mean of elements
<code>np.std</code>	<code>np.nanstd</code>	Compute standard deviation
<code>np.var</code>	<code>np.nanvar</code>	Compute standard deviation
<code>np.min</code>	<code>np.nanmin</code>	Find minimum value
<code>np.max</code>	<code>np.nanmax</code>	Find maximum value
<code>np.argmin</code>	<code>np.nanargmin</code>	Find index of minimum value
<code>np.argmax</code>	<code>np.nanargmax</code>	Find index of maximum value
<code>np.median</code>	<code>np.nanmedian</code>	Compute median of elements
<code>np.percentile</code>	<code>np.nanpercentile</code>	Compute rank-based statistics of elements
<code>np.any</code>	N/A	Evaluate whether any elements are true
<code>np.all</code>	N/A	Evaluate whether all elements are true

# NumPy

## Universal functions: Aggregations (II)

(Download dataset)

- Use `wget` or `curl` to download the file within `iPython`

Basic data analysis example

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

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

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



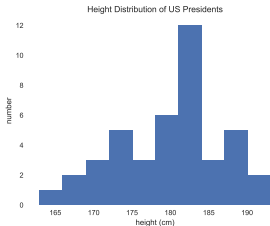
# NumPy

## Universal functions: Aggregations (III)

### Basic data analysis example (Continuation)

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

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



# NumPy

## 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.

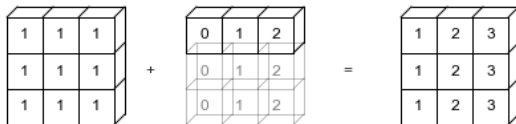
# NumPy

## Universal functions: Broadcasting (II)

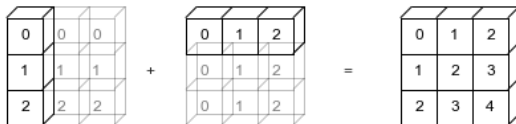
`np.arange(3)+5`



`np.ones((3,3))+np.arange(3)`



`np.arange(3).reshape((3,1))+np.arange(3)`



Array expansion does not  
consume memory!

# NumPy

## Universal functions: Broadcasting (III)

### Normalization

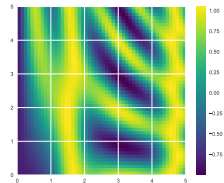
```
X = np.random.random((10, 3))
Xmean = X.mean(0)
X_centered = X - Xmean
```

### 3D plot

```
%matplotlib inline
import matplotlib.pyplot as plt

x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[: , np.newaxis]
z = np.sin(x)**10 + np.cos(10 + y*x)*np.cos(x)

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



# NumPy

## Comparisons, masks, and Boolean logic (I)

(Download dataset)

### Example

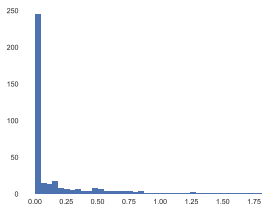
```
import numpy as np
import pandas as pd

# pandas to extract rainfall inches as a ndarray
rainfall = pd.read_csv('Seattle2014.csv')['PRCP'].values
inches = rainfall / 254.0 # 1/10mm -> inches
inches.shape
# Outputs (365,)
```

```
%matplotlib
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
plt.hist(inches, 40);
```

# NumPy

## Comparisons, masks, and Boolean logic (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

# NumPy

## Comparisons, masks, and Boolean logic: Booleans arrays masks (I)

### Syntax examples

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

We've seen arithmetic ufuncs ...

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

OPERATOR	UFUNC
<code>==</code>	<code>np.equal</code>
<code>!=</code>	<code>np.not_equal</code>
<code>&lt;</code>	<code>np.less</code>
<code>&lt;=</code>	<code>np.less_equal</code>
<code>&gt;</code>	<code>np.greater</code>
<code>&gt;=</code>	<code>np.greater_equal</code>

OPERATOR	UFUNC
<code>&amp;</code>	<code>np.bitwise_and</code>
<code> </code>	<code>np.bitwise_or</code>
<code>^</code>	<code>np.bitwise_xor</code>
<code>~</code>	<code>np.bitwise_not</code>

# NumPy

## Comparisons, masks, and Boolean logic: Booleans arrays masks (II)

### Example

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

np.sum(~((inches <= 5) | (inches >= 1)))
```



## NumPy

## 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

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

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

## NumPy

### Structured arrays

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

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

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

# Pandas

## XXX

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

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

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