Scientific Programming in Python

Inteligencia Artificial en los Sistemas de Control Autónomo
Máster en Ciencia y Tecnología desde el Espacio

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.

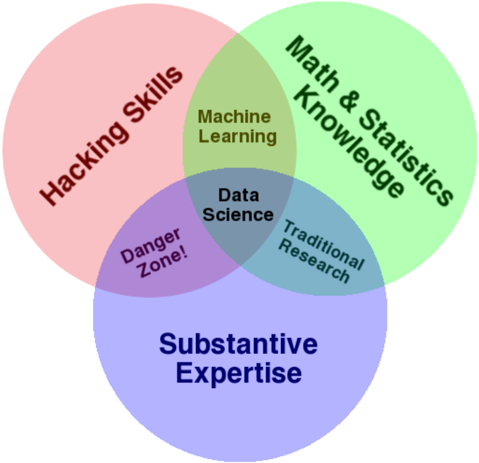
Bibliography

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

Table of Contents

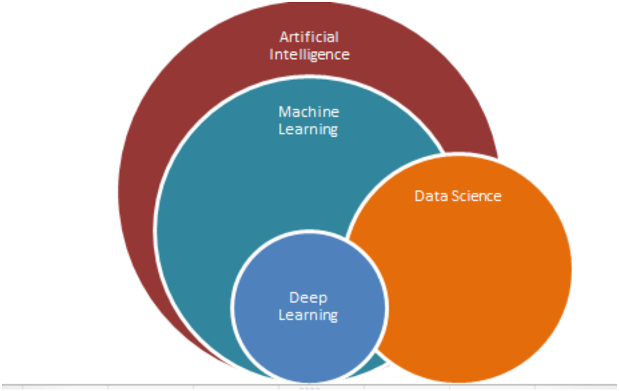
Overview

Data Science (I)



Overview

Data Science (II)



(Source)

Data Science

The data scientist tookit (I)

Data science is about manipulating data

- Need of specialized tools
- Two main languajes: 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 tookit (II)

Tool	Type	Description
`conda`	Software	Python environments and package management
`iPython`	Software	Advaced Python interpreter
`Jupiter`	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
- Environment management for Python
- Package management system

Anaconda provides `conda`

- Packages management tool
- Environment management for Python

In addition, Anaconda provides `Spyder`

- Python IDE designed for Data Science

Data Science

Conda crush introduction

Conda environment for Data Science

1. `conda create --name ml seaborn=0.9.0`
2. `source activate ml`
3. `conda install ipython`
4. `conda install nb_conda`

List environments:

```
conda info --envs
```

Activate environment:

```
source activate <env>
```

Install package:

```
conda install <package>
```

List packages:

```
conda list
```

Exit environment (Linux):

```
deactivate
```

Data Science

Python IDEs for Data Science (I)

iPython

iPython = Interactive Python

- Extended funcionality
- Enhanced UI
- External editor

Running iPython:

`$ ipython`

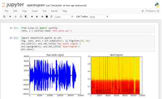
Jupyter

Python notebooks

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

Running Jupyter:

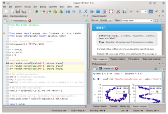
`$ jupyter notebook`



Spyder

Matlab-like IDE

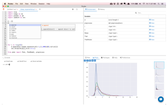
- Default IDE in Anaconda
- Uses iPython



Rodeo

Python version of RStudio

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



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. Jupyter.
   - Bonus track: Publish the notebook in GitHub.
3. Spyder.
4. Rodeo (optional).

iPython

Basics (I)

In regular Python ...

- most objects come with a docstring attribute
- docstring accesible thorugh `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 buildin 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
                    numba      numpy
                    numbers    numpydoc
                    numexpr
```

- 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 handly 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

Pasting code blocks: %paste and %cpaste

Pasting code in Python is troublesome

- %paste: Paste one time
- % %cpaste: Paste several times

```
def donothing(x):
    return x
```

%paste

```
In [20]: %paste
   def donothing(x):
    return x

## -- End pasted text --
```

%cpaste

```
In [25]: %cpaste
Pasting code; enter '--' alone on the line
to stop or use Ctrl-D.
:       def donothing(x):
            return x:
:--
```

iPython

Running external code: %run and %timeit

`%run`: Execute script

- Many optional arguments
- Checkout `%run?`

```
In [40]: %run donothing.py

In [41]: donothing(10)
Out[41]: 10
```

`%timeit`: Computes execution time

- Executes a single line
- Automatic adjustment of runs
- Shows basic statistics

```
In [33]: %timeit [n ** 2 for n in range(200)]
71.6 µs ± 1.84 µs per loop
(mean ± std. dev. of 7 runs, 10000 loops each)

In [34]: %timeit [n ** 2 for n in range(2000)]
753 µs ± 16.2 µs per loop
(mean ± std. dev. of 7 runs, 1000 loops each)
```

`%%timeit`: Several lines

iPython

Input and output history (I)

iPython stores its history as objects

- `In`: Input commands
  - List storing commands
- `Out`: Commands output
  - Dictionary storing outputs
  - Not all commands have outputs

```
In [1]: import math
In [2]: math.sin(2)
Out[2]: 0.9092974268256817
In [3]: math.cos(2)
Out[3]: -0.4161468365471424
In [4]: Out[2] ** 2 + Out[3] ** 2
Out[4]: 1.0
```

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, 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)

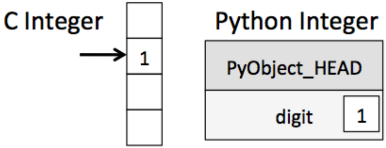
NumPy

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];
};
```

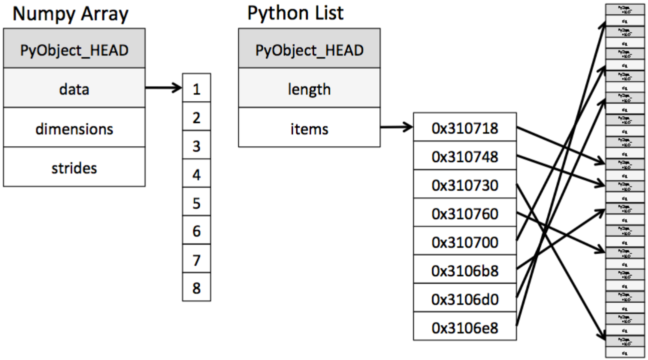


NumPy

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
- Homogeneus 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
`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

NumPy

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("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
print("dtype:", x3.dtype)
print("itemsize:", x3.itemsize)
print("nbytes:", x3.nbytes)
```

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 truncaded 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 a rray

```
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)
# 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
```

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 advantange 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
`+`	`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)

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.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)`

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
`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

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))
```

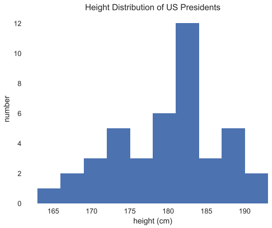
NumPy

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

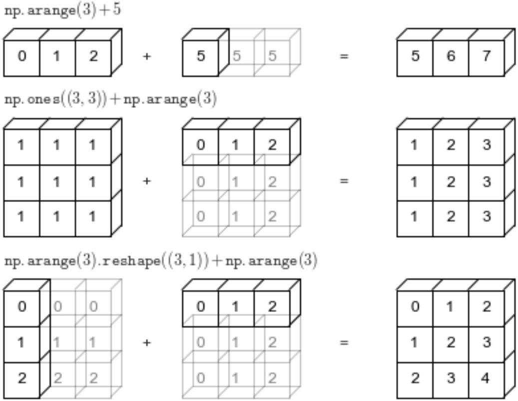
- Used in ufuncs
- Implicit array expansion through three rules

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)



Array expansion does not consume memory!

NumPy

Universal functions: Broadcasting (III)

Normalization

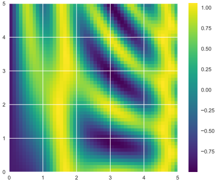
```
X = np.random.random((10, 3))
Xmean = X.mean(0)
X_centered = X - Xmean
```

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

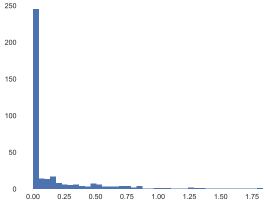
```
import numpy as np
import pandas as pd

# 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
==	`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`

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

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

NumPy

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

```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
```

Solution: Structured arrays

Structured arrays

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

NumPy

Structured arrays (II)

Structured array manipulation

```
data['name'] = name
data['age'] = age
data['weight'] = weight

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

These kind of structures are day-to-day used

- Pandas is a much better choice

Pandas

Introduction

A data science workflow needs more features

- Label columns and rows
- Missing data
- Operations on groups
- Data input

Pandas implements all those features, and more

- Built on NumPy's ndarray

Pandas provides two main objects

- `Series`
- `DataFrame`

Convention

```
import numpy as np
import pandas as pd
```

Pandas

Introduction

A DS/ML workflow needs more features

- Missing data
- Data input
- Operations on groups
- Label columns and rows

Pandas provides all those features, and more

- Pandas = **PAN**el **DA**ta **S**ystem
- Built on NumPy's ndarray
- Provides dataframes

Pandas provides two main objects

- `Series` and `DataFrame`

Convention

```
import numpy as np
import pandas as pd
```

Pandas

The Pandas `Series` object (I)

A `Series` is a one-dimensional array of indexed data

- NumPy arrays indices are implicit (i.e. its position)
- Series indices are explicit, and can be any type

Two attributes

- `values`: ndarray
- `index`: `pd.Index` object

Two indices

- Implicit: Regular index
- Explicit: Custom index

Index	Values
'a'	0.25
'b'	0.5
'c'	0.75
'd'	0.99

```
data = pd.Series([0.25,
    0.5, 0.75, 1.0])
data.values
data.index
data[1:3]
```

Pandas

The Pandas Series object (II)

Custom indices

```
In[1]  : data = pd.Series([0.25, 0.5, 0.75, 1.0],
                  index=['a', 'b', 'c', 'd'])
In [2]: data
Out[1]:
a      0.25
b      0.50
c      0.75
d      1.00
dtype: float64

In [3]: data['a']
Out[2]: 0.25
In [4]: data[0]
Out[3]: 0.25
```

Pandas

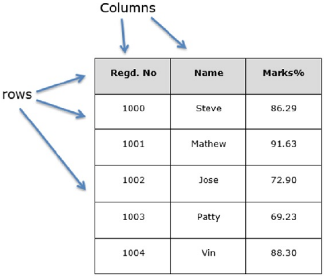
The Pandas `DataFrame` object (I)

A `DataFrame` is a 2-D tabular data structure

- Similar to a spreadsheet
- Homogeneous columns
- Heterogeneous rows

Two read-only attributes, both `pd.Index`

- `index`: Rows
- `columns`: Columns



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

(Source)

Pandas

The Pandas DataFrame object (II)

DataFrame example

```
In [1]:  import seaborn as sns

In [2]:  iris = sns.load_dataset('iris')
In [3]:  iris.head()

Out[1]:
sepal_length  sepal_width  petal_length  petal_width  species
0           5.1          3.5           1.4          0.2   setosa
1           4.9          3.0           1.4          0.2   setosa
2           4.7          3.2           1.3          0.2   setosa
3           4.6          3.1           1.5          0.2   setosa
4           5.0          3.6           1.4          0.2   setosa
In [246]: iris.columns
Out[246]:
Index(['sepal_length', 'sepal_width', 'petal_length',
       'petal_width', 'species'], dtype='object')
```

Pandas

Constructing DataFrame objects

Manual initialization

- From a single `Series` object
  `pd.DataFrame(population, columns=['population'])`
- From several `Series` objects
  `pd.DataFrame('population': population, 'area': area)`
- From a dictionary
  `pd.DataFrame([{'a': 0, 'b': 0}, {'a': 1, 'b': 2}])`
- From a NumPy 2-D array
  `pd.DataFrame(np.random.rand(3, 2),`
  `columns=['foo', 'bar'], index=['a', 'b', 'c'])`

Read from a file

- CSV (very common!!!): `pd.read_csv('filename.csv')`
- Excel:
  `pd.read_excel('filename.xlsx', sheetname='mysheet')`

Pandas

Data indexing and selection: Series

Dictionary-like syntax

```
>>> data = pd.Series([0.25, 0.5,
    0.75, 1.0], index=['a', 'b
    ', 'c', 'd'])

>>> 'a' in data
True

>>> data.keys()
Index(['a', 'b', 'c'], dtype='
    object')

>>> list(data.items())
[('a', 0.25), ('b', 0.5), ('c',
    0.75)]

>>> data['e'] = 1.25
```

Array-like syntax

```
>> data['a':'c'] #Explicit index
a     0.25
b     0.50
c     0.75
dtype: float64
>> data[0:2] # Implicit index
a     0.25
b     0.50
dtype: float64
>> data[data>0.5] # Masking
c     0.75
d     1.00
dtype: float64
>> data[['b','c']] # Fancy index
b     0.50
c     0.75
dtype: float64
```

Pandas

Data indexing and selection: DataFrame

Dictionary-like syntax

```
>>> data['area']
>>> data.area
>>> data.area is data['area']
True
>>> data['density'] = data['pop'
    ]/data['area']
```

Array-like syntax

```
>>> data.values # Get values
    array
>>> data.T # Transpose
>>> data[0] # First row
>>> data['area'] # Area column
```

Remember indexing conventions

- Indexing refers to columns (`data['area']`)
- Slicing refers to rows (`data['Florida':'Illinois']`)
- Masking refers to rows (`data[data.density > 100]`)

Pandas

Data indexing and selection: loc, iloc and ix

Two types of indices in Pandas

- Explicit and implicit
- Indexing (`data[0]`) is explit
- Slicing (`data[:2]`) is implicit (Python-like)
- Source of troubles!

Pandas makes explicit the used scheme

- `loc`: Explicit index
- `iloc`: Implicit index
- `ix`: Hybrid

```
# Series
>>> serie.loc[1]
>>> serie.loc[1:3]
>>> serie.iloc[1]
>>> serie.iloc[1:3]

# Dataframes
>>> df.iloc[:3, :2]
>>> df.loc[:'illinois', :'pop']
>>> df.ix[:3, :'pop']
>>> df.loc[df.data>100, ['pop',
    'density']]
>>> df.iloc[0, 2] = 90
```

Pandas

Operating on data (I)

Pandas fully supports NumPy's ufuncs

- Efficient computations

Additional Pandas features

- Index and column name preservation
- Index aligning
- Easy data combination

```
>>> rng = np.random.RandomState(42)
>>> df = pd.DataFrame(rng.randint(0,
    10, (3,4)))
>>> df = pd.DataFrame(rng.randint(0,
    10, (3,4)), columns=['A', 'B', 'C'
    , 'D'])
>>> print(df)
   A  B  C  D
0  7  2  5  4
1  1  7  5  1
2  4  0  9  5
>>> np.sin(df * np.pi / 4)
          A      B     C         D
 0 -7.07e-01   1.0  -0.7   1.22e-16
 1  7.07e-01  -0.7  -0.7   7.07e-01
 2  1.22e-16   0.0   0.7  -7.07e-01
```

Pandas

Operating on data (II)

Index preservation

```
>>> A = pd.Series([2, 4, 6], index=[0, 1, 2])
>>> B = pd.Series([1, 3, 5], index=[1, 2, 3])
>>> A + B
0    NaN
1    5.0
2    9.0
3    NaN
dtype: float64
>>> A.add(B, fill_value=0)
0    2.0
1    5.0
2    9.0
3    5.0
dtype: float64
```

Pandas

Missing data (I)

NumPy supports missing data in floating-point data

- Specific value defined by IEEE
- Available as `np.nan`

Pandas supports missing data through two mechanisms

- `None` object, interpreted as NaN (`Not a Number`)
- `np.nan`: for floating-point data
- Almost automatic NaN handling (types upcast)

```
>>> pd.Series([1, np.nan, 2, None])
0     1.0
1     NaN
2     2.0
3     NaN
dtype: float64
```

Pandas

Missing data (II)

Useful functions for missing data

- `isnull()`: Boolean mask with missing data
- `notnull()`: Opposite of isnull()
- `dropna()`: Filtered data
- `fillna()`: NaNs filled

```
>>> data = pd.Series([1, np.nan,
      'hello', None])
>>> data[data.notnull()]
0        1
2    hello
dtype: object

>>> data.dropna()
0        1
2    hello
dtype: object

>>> data.fillna(0)
0        1
1        0
2    hello
3        0
dtype: object
```

Pandas

Combining datasets: `pd.concat()` (I)

Many times we need to combine two or more datasets

- Pandas provides `pd.concat()`, `append()` and `pd.merge()`

pd.concat() signature

```
pd.concat(objs, axis=0, join='outer',
    join_axes=None, ignore_index=False, keys
    =None, levels=None, names=None,
    verify_integrity=False, copy=True)
```

By default, `pd.concat()` joins rows preserving index

- `axis`: Join columns (`axis=1`)
- `verify_integrity`: Raise error if duplicates (`verify_integrity=True`)
- `ignore_index`: Create new index (`ignore_index=True`)
- `join`: Can be 'outer' (union) or 'inner' (intersection)

Pandas

Combining datasets: pd.concat() (II)

```
>> df1 = pd.DataFrame([{'A': 'A0', 'B': 'B0'}, {'A': 'A1', 'B': 'B1'
    }])
>> df2 = pd.DataFrame([{'A': 'A2', 'B': 'B2'}, {'A': 'A3', 'B': 'B3'
    }])

>> print(df1), print(df2); print(pd.concat([df1, df2]))
   A   B         A   B        A   B
0  A0  B0    0  A2  B2    0  A0  B0
1  A1  B1    1  A3  B3    1  A1  B1
                          0  A2  B2
                          1  A3  B3
>> pd.concat([df1, df2], axis=1)
   A   B   A   B
0  A0  B0  A2  B2
1  A1  B1  A3  B3
>> df1.append(df2)
```

Pandas

Combining datasets: `pd.merge()` (I)

Merging based on relational algebra

- Similar to databases tables joins
- Pretty intelligent figuring out the desired output
- By default, join dataframes using shared columns names

Pandas

Combining datasets: pd.merge() (II)

One-to-one

```
>> print(df1); print(df2)
 employee        group
0      Bob  Accounting
1     Jake Engineering
2     Lisa Engineering
3      Sue          HR
 employee  hire_date
0     Lisa      2004
1      Bob      2008
2     Jake      2012
3      Sue      2014
>> print(pd.merge(df1, df2))
 employee group  hire_date
0      Bob Accounting   2008
1     Jake Engineering  2012
2     Lisa Engineering  2004
3      Sue HR           2014
```

Many-to-one

```
>>> print(df3); print(df4)
  employee   group  hire_date
0      Bob  Accounting   2008
1     Jake Engineering   2012
2     Lisa Engineering   2004
3      Sue          HR   2014
        group  supervisor
0  Accounting  Carly
1 Engineering  Guido
2          HR  Steve
>> print(pd.merge(df3, df4))
employee   group  hire_date  supervisor
0 Bob     Accounting   2008  Carly
1 Jake   Engineering   2012  Guido
2 Lisa   Engineering   2004  Guido
3 Sue             HR   2014  Steve
```

Pandas

Combining datasets: `pd.merge()` (III)

Many-to-many

```
>>> print(df1); print(df5)
  employee        group              group        skills
0      Bob   Accounting     0   Accounting          math
1     Jake  Engineering     1   Accounting  spreadsheets
2     Lisa  Engineering     2  Engineering        coding
3      Sue           HR     3  Engineering         linux
                            4           HR  spreadsheets
                            5           HR  organization
>>> pd.merge(df1, df5)
  employee        group        skills
0      Bob   Accounting          math
1      Bob   Accounting  spreadsheets
2     Jake  Engineering        coding
3     Jake  Engineering         linux
4     Lisa  Engineering        coding
5     Lisa  Engineering         linux
6      Sue           HR  spreadsheets
7      Sue           HR  organization
```

Pandas

Combining datasets: `pd.merge()` (IV)

pd.merge() signature

```
pd.merge(left, right, how='inner', on=None,
    left_on=None, right_on=None, left_index=
    False, right_index=False, sort=False,
    suffixes=('_x', '_y'), copy=True,
    indicator=False, validate=None)
```

Arguments:

- `on`: Key column name
- `left_on`: Left table key column name
- `right_on`: Right table key column name
- `how`: Set arithmetic, `'inner'` (default, intersection), `'outer'` (union, fills missings with NaNs), `'left'` (left entries), `'right'` (right entries)

Pandas

Combining datasets: pd.merge() (V)

```
>>> A                   >>> B
    lkey  value             rkey  value
0   foo   1             0   foo   5
1   bar   2             1   bar   6
2   baz   3             2   qux   7
3   foo   4             3   bar   8

>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
    lkey  value_x  rkey  value_y
0   foo   1        foo   5
1   foo   4        foo   5
2   bar   2        bar   6
3   bar   2        bar   8
4   baz   3        NaN   NaN
5   NaN   NaN      qux   7
```

Pandas

Aggregation in Pandas (I)

The first step in data analysis is summarization

- First contact with data
- Insight to the dataset

Aggregation methods

- Applied to columns

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard dev. and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items
`describe()`	Data summary

```
>>> import seaborn as sns
>>> planets = sns.load_dataset('planets')
>>> planets.head()
          method  number  orbital_period   mass  distance  year
0  Radial Velocity  1       269.300         7.10     77.40  2006
1  Radial Velocity  1       874.774         2.21     56.95  2008
2  Radial Velocity  1       763.000         2.60     19.84  2011
3  Radial Velocity  1       326.030        19.40    110.62  2007
4  Radial Velocity  1       516.220        10.50    119.47  2009
>>> planets.dropna().describe()
       number  orbital_period    mass  distance      year
count  498.00      498.000000  498.00  498.0000   498.000
mean     1.73      835.778671    2.50   52.0682  2007.377
std      1.17     1469.128259    3.63   46.5960     4.167
min      1.00        1.328300    0.00    1.3500  1989.000
25%      1.00       38.272250    0.21   24.4975  2005.000
50%      1.00      357.000000    1.24   39.9400  2009.000
75%      2.00      999.600000    2.86   59.3325  2011.000
max      6.00    17337.500000   25.00  354.0000  2014.000
>>> planets.mean()
number               1.785507
orbital_period    2002.917596
mass                 2.638161
distance           264.069282
year              2009.070531
dtype: float64
```

Pandas

Grouping in Pandas (I)

Aggregation is generally used ...

- ... good to operate with the whole dataset ...
- ... but also is is usually insufficient

We need conditional aggregations

- Aggregate conditionally on some label

This is done with the operation `groupby` (yes, that name comes from SQL)

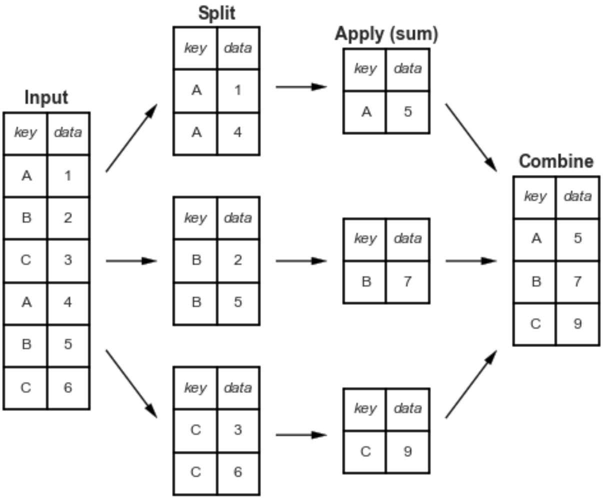
- Example: `df.groupby("key")`

Three tasks in one step

1. *Split*: Break up dependening on a key
2. *Apply*: Compute some function
3. *Combine*: Merge results into an output

Pandas

Grouping in Pandas (II)



Pandas

Grouping in Pandas (III)

```
>>> df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                       'data': range(6)})
>>> print(df)
  key  data
0   A     0
1   B     1
2   C     2
3   A     3
4   B     4
5   C     5
>>> df.groupby('key')
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0
    x102685438>
>>> df.groupby('key').sum()
     data
key
A       3
B       5
C       7
```

Pandas

Grouping in Pandas (III)

Several mapping methods available

- List
  `df.groupby([2,3,4,1]).sum()`
- Dictionary
  `df.groupby('A': 'vowel', 'B': 'consonant', 'C': 'vowel')`
- Python function
  `df.groupby(str.lower)`
- Multiple keys
  `planets.groupby(['method', 'year'])`
- Mixed keys
  `df.groupby(['key1', 'key2', str.lower])`

Pandas

Grouping in Pandas (IV)

The method `groupby()` returns an object `groupby`

- Basicly, it is a collection of dataframes
  `planets.groupby('method').get_group('Transit')`
- Column selection as dataframe
  `planets.groupby('method')['year']`

Interesting groupby attribute, `groups`

- Dictionary with groups
  `planets.groupby('method').groups`
- Compatible with the `len()` method
  `len(planets.groupby('method'))`

Pandas

Grouping in Pandas (V)

Usual operations with groupings

- Aggregation:
  `df.groupby('key').aggregate(['min', np.median, max])`
  `df.groupby('key').aggregate('data1': 'min', 'data2': 'max')`
- Filtering:
  `planets.groupby('method').filter(lambda x: x['distance'].mean() > 50.)`
- Transformation:
  `df.groupby('key').transform(lambda x: x - x.mean())`

Apply(): Apply arbitrary function and combine results

- Takes a function as argument that takes a DataFrame
  `planets.groupby("method").apply(lambda x: x / x.sum())`

Pandas

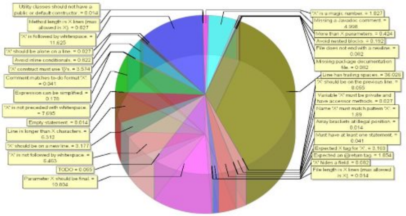
Grouping in Pandas (VI)

Grouping by decade

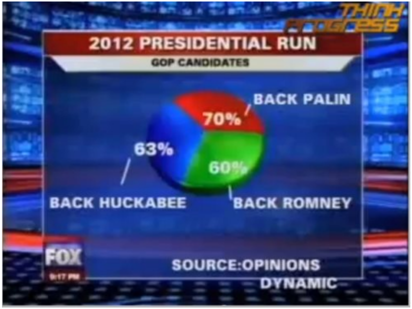
```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])['number'].sum()
    .unstack().fillna(0)
```

Visualization

Bad visualization examples (I)



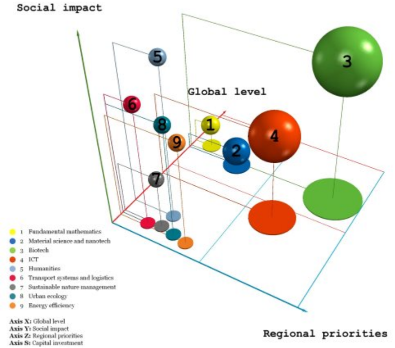
(Source)



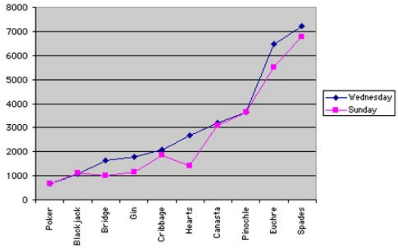
(Source)

Visualization

Bad visualization examples (II)



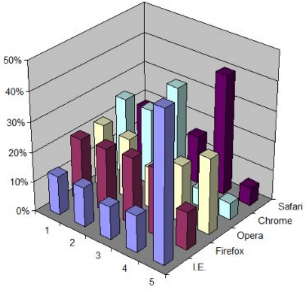
(Source)



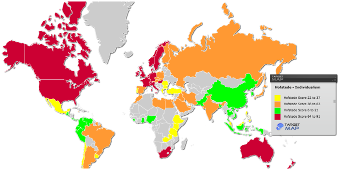
(Source)

Visualization

Bad visualization examples (III)



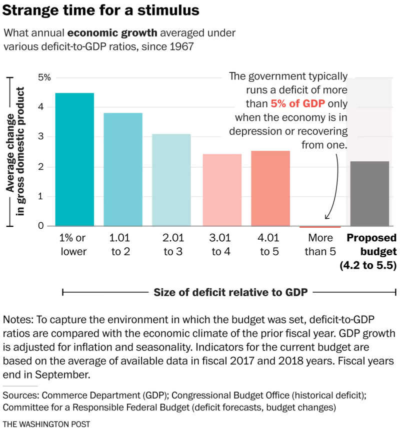
(Source)



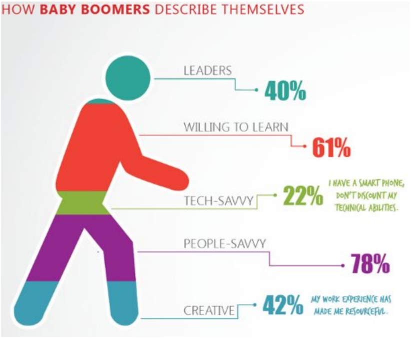
(Source)

Visualization

Bad visualization examples (IV)



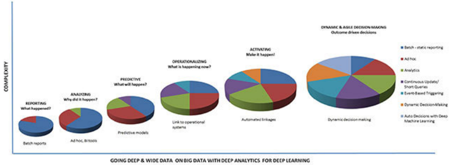
(Source)



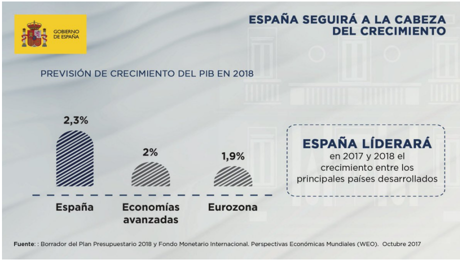
(Source)

Visualization

Bad visualization examples (V)



(Source)



(Source)

Visualization

Motivation (I)

Efficient data visualization tips

- **Define your story**
- The chart must tell the story
- Don't distract from your story (with irrelevant data or visual elements)
- One story, one chart
- Put the story comprension in first term
- Better several simple charts than one complex chart
- Choose colors wisely (color scale or high contrast)
- Elements order must support the story (leyend, bars, etc)
- There is life beyond pies and bars
- Keep it simple, stupid!

Visualization

Motivation (II)

Know your data

- Categorical or numerical
- Number of dimensions to represent (1D, 2D, 3D, more dimensions)

Can you use other representation?

- Chart better than table? ...
- ... that depends

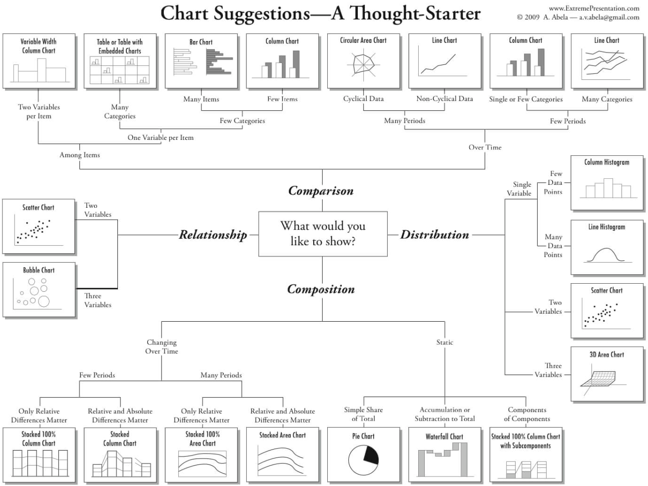
What do you want to represent?

- Distribution, relationship, comparison or composition

Look for templates: (`https://python-graph-gallery.com/`)

Visualization

Motivation (III)



(Source)

(Alternative resource)

Visualization

Introduction to Matplotlib (I)

Matplotlib is a Python package

- Based on NumPy
- Imitates Matlab

Three operation modes

- Scripts.
  Must use `plt.show()` to enter event loop. Use it once!
- IPython shell.
  Must use `%matplotlib`
- IPython notebook. Two modes
  - `%matplotlib inline`
  - `%matplotlib notebook`

Convention

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

myplot.py

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.show()
```

Visualization

Introduction to Matplotlib (II)

Matplotlib comes with two interfaces

- Matlab-like. Old-fashioned function-oriented API.
- Object-oriented. Object-oriented and more powerfull API.

Matlab API

```
plt.figure()  # create a plot

# create the first of two panels
    and set current axis
plt.subplot(2, 1, 1) # (rows,
    columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel and
    set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```

OO API

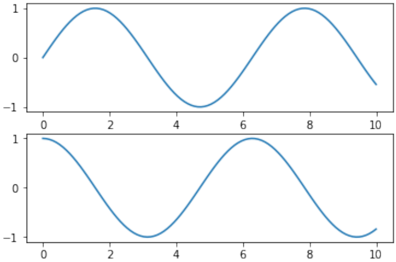
```
plt.figure()  # create a plot

# create the first of two panels
    and set current axis
plt.subplot(2, 1, 1) # (rows,
    columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel and
    set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```

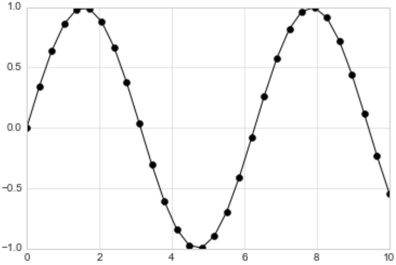
Visualization

Introduction to Matplotlib (III)

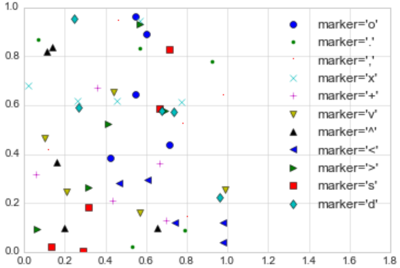


Visualization

Introduction to Matplotlib (IV)



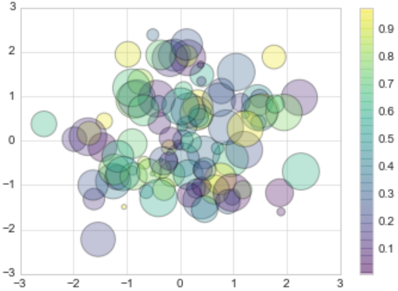

```
plt.plot(x, np.sin(x), '-ok',
     color='black')
```

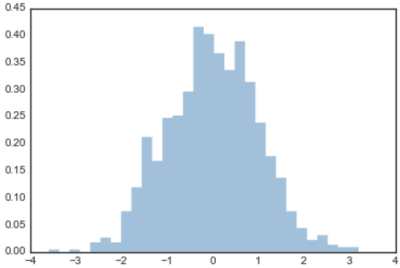



```
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '
    <', '>', 's', 'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker,
         label="marker='{0}'".format(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```

Visualization

Introduction to Matplotlib (V)





```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
            cmap='viridis')
plt.colorbar(); # show color scale
```

```
data = np.random.randn(1000)

plt.hist(data, bins=30, normed=True, alpha=0.5,
         histtype='stepfilled', color='steelblue',
         edgecolor='none');
```

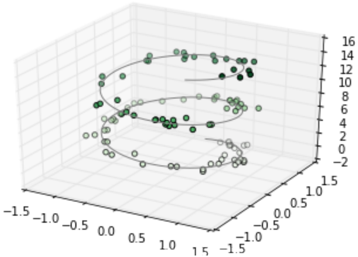
Visualization

Introduction to Matplotlib (VI)

```
ax = plt.axes(projection='3d')

# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')

# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='
      Greens');
```



Visualization

Introduction to Seaborn (I)

Seaborn is a modern data-visualization Python package

- Based on matplotlib
- ... it uses matplotlib indeed
- Pandas-aware
- High level
- Advanced visualizations
- Easy to use

Still under development! (v. 0.9)

Convention

```
import seaborn as sns
```

This documentation is for Seaborn 0.9 or newer

Visualization

Introduction to Seaborn (II)

Display initialization

- `plt.show()`
- `%matplotlib`

Style initialization

- Default Seaborn style `sns.set()`
- By default, same style than matplotlib

Several functions ...

- ... similar parameters

Parameters

- x: Data axis x
- y: Data axis Y
- data: Dataframe name
- hue: Color
- style: Style
- sizes: Size
- kind: Alternate representation

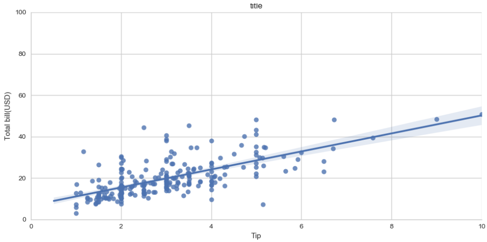
Visualization

Introduction to Seaborn (III)

Typical Seaborn usage

1. Prepare data
2. Set up aesthetics
3. Plot
4. Customize the plot

```
import matplotlib.pyplot as plt
import seaborn as sns
# Prepare data
tips = sns.load_dataset("tips")
# Set up aesthetics
sns.set_style("whitegrid")
# Plot
g = sns.lmplot(x="tip",y="total_bill", data=tips,aspect=2)
# Plot customization
g = (g.set_axis_labels("Tip","Total bill(USD)").set(xlim
      =(0,10),ylim=(0,100)))
plt.title("title")
plt.show(g)
```



Visualization

Seaborn datasets (I)

Seaborn comes with several dummy datasets

- `sns.load_dataset('name')`

We will use two datasets

- 'iris': The classical iris dataset, numerical
- 'tips': Numeric and categorical variables

Tips dataset

```
>>> tips = sns.load_dataset('tips')
>>> print(tips.head())
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Visualization

Seaborn datasets (II)

Iris dataset

```
>>> iris = sns.load_dataset('iris')
>>> print(iris.head())
```

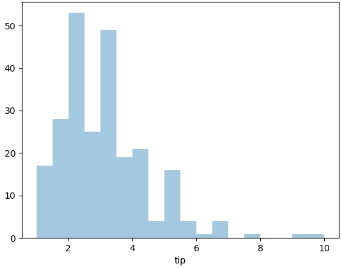
	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

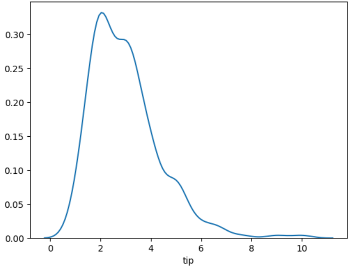
Iris Versicolor **Iris Setosa** **Iris Virginica**

(Source)

Visualization

Seaborn: Distributions (I)





Histogram

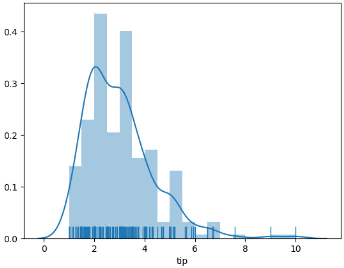
```
sns.distplot(tips['tip'],
    kde=False)
```

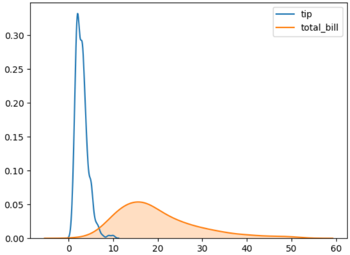
Density plot

```
sns.distplot(tips['tip'],
    hist=False)
```

Visualization

Seaborn: Distributions (II)





Histogram + density plot

```
sns.distplot(tips['tip'],
    rug=True)
```

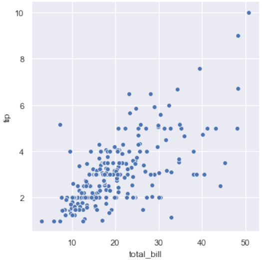
Density plot

```
sns.kdeplot(tips['tip'])
sns.kdeplot(tips['total_bill
    '], shade=True)
```

Visualization

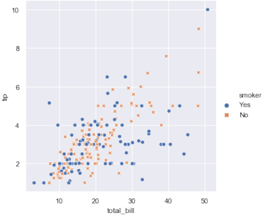
Seaborn: Relationships (I)

Scatterplots

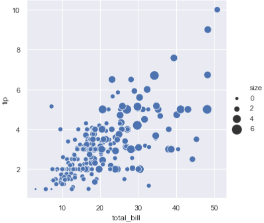



```
sns.relplot(x="total_bill", y="tip", data=tips)
```

Seaborn >= 0.9



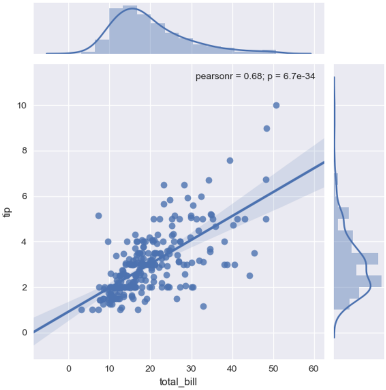

```
sns.relplot(x="total_bill", y="tip", hue="smoker", style="smoker", data=tips)
```

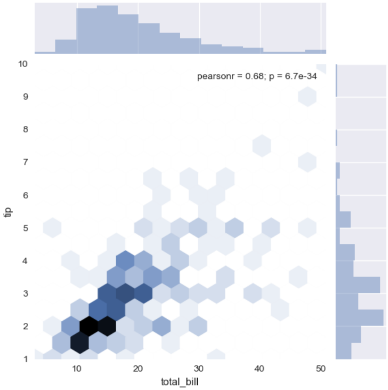



```
sns.relplot(x="total_bill", y="tip", size="size", sizes=(15, 200), data=tips);
```

Visualization

Seaborn: Relationships (II)



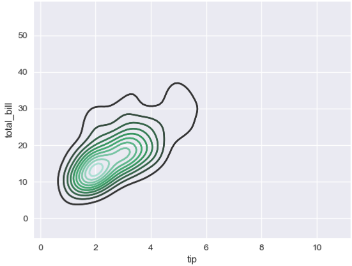



```
sns.jointplot("total_bill", "tip", tips, kind="reg"
)
```

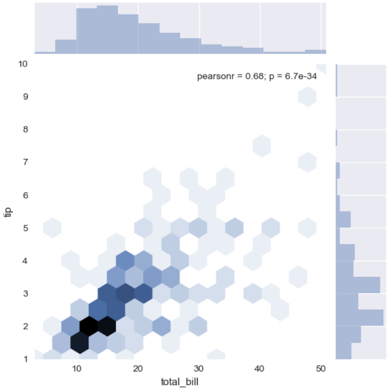
```
sns.jointplot("total_bill", "tip", tips , kind="hex
")
```

Visualization

Seaborn: Relationships (III)



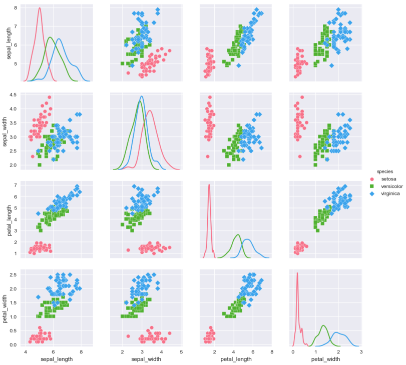

```
sns.kdeplot(tips['tip'], tips['total_bill'])
```

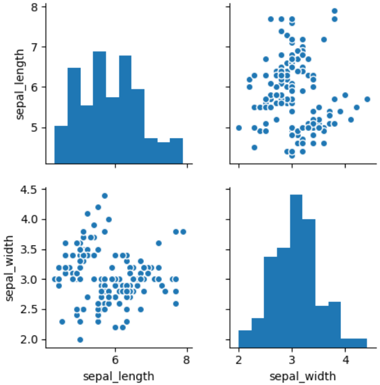



```
sns.jointplot("total_bill", "tip", tips , kind="hex
")
```

Visualization

Seaborn: Relationships (IV)





Scatterplot matrix

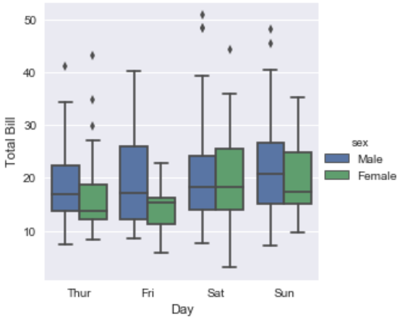
```
sns.pairplot(iris, hue="species", palette="husl",
        markers=["o", "s", "D"], diag_kind='kde')
```

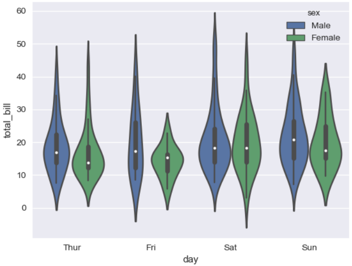
Scatterplot matrix

```
sns.pairplot(iris, vars=["sepal_length", "
    sepal_width"])
```

Visualization

Seaborn: Comparisons (I)





Boxplot

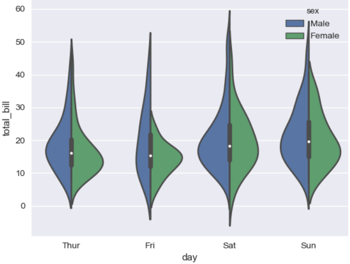
```
with sns.axes_style(style='ticks'):
    g = sns.factorplot("day", "total_bill", "sex",
        data=tips, kind="box")
g.set_axis_labels("Day", "Total Bill")
```

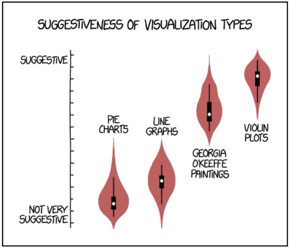
Violin plot

```
sns.violinplot("day", "total_bill", "sex", data=
    tips)
```

Visualization

Seaborn: Comparisons (II)





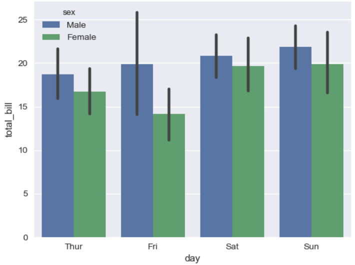
(Source)

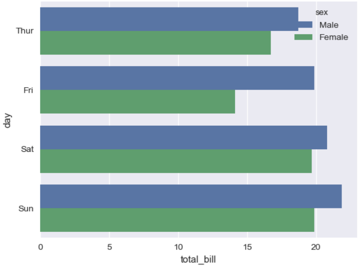
Violin plot

```
sns.violinplot(x="day", y="total_bill", hue="sex",
        data=tips, split=True)
```

Visualization

Seaborn: Barplots





Barplot

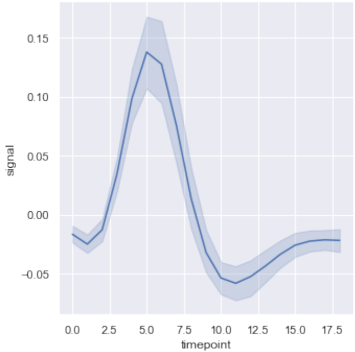
```
sns.barplot(x="day", y="total_bill", hue="sex",
        data=tips)
```

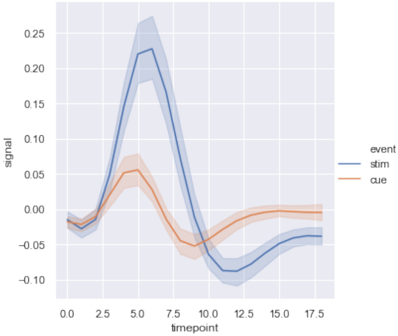
Barplot

```
sns.barplot(x="total_bill", y="day", hue="sex",
        data=tips, ci=None)
```

Visualization

Seaborn: Continuity





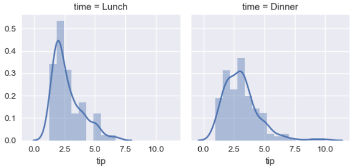

```
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint", y="signal", kind="line",
        data=fmri)
```

```
sns.relplot(x="timepoint", y="signal", hue="event",
        kind="line", data=fmri)
```

Seaborn >= 0.9

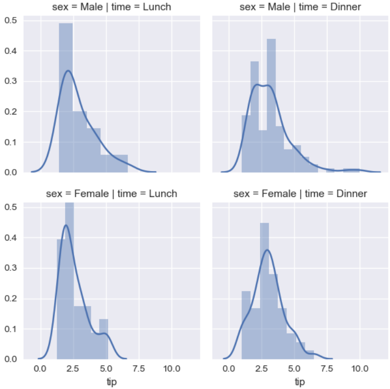
Visualization

Seaborn: FacetGrid

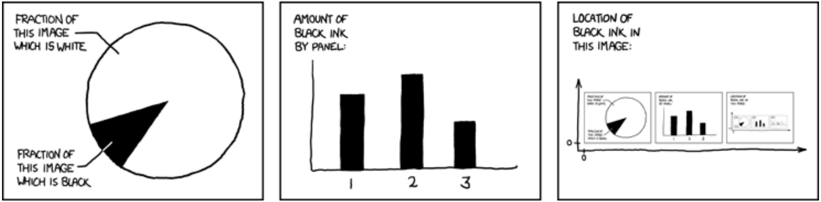



```
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint", y="signal", kind="line",
        data=fmri)
```

Seaborn >= 0.9




```
g = sns.FacetGrid(tips, col="time", row="sex")
g.map(sns.distplot, "tip")
```


FRACTION OF THIS IMAGE WHICH IS WHITE
FRACTION OF THIS IMAGE WHICH IS BLACK
AMOUNT OF BLACK INK BY PANEL:
1
2
3
LOCATION OF BLACK INK IN THIS IMAGE:
0
0

(Source)