# 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

#### I. 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

#### J..... D...

#### 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

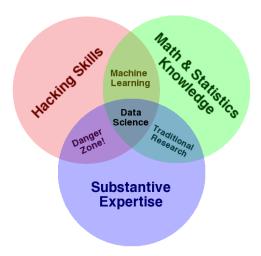
#### 4. Pandas

- Introduction
  - The Pandas Series object
  - The Pandas DataFrame object
  - Constructing DataFrame objects
  - = D · · 1 · · · 1 · · ·
  - Data indexing and selectionOperating on data
  - Mr. . 1.
  - Missing data
  - Combining datasets: pd. concat()
  - Combining datasets: pd.merge()
- Aggregation in Pandas
- Grouping in Pandas

#### 5. Visualization

- Visualization examples
- Introduction to Matplotlib
- Introduction to Seaborn
- Seaborn: Distribution





## 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



### The data scientist tookit (II)

Tool	Туре	Description
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

#### Anaconda

Overview

#### 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







Overview

## 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





#### 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.



## 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 `?'

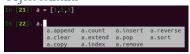
Easy access to source code with `??'

Does not work with most buildin functions!



## Press <tab> to complete almost everything

Object contents



Packages



Wildcards





## $i \\ Python$

## 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 $(\downarrow)$	Next command
Ctrl-r	Reverse-search

#### Text entry

1	
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 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



## Pasting code blocks: %paste and %cpaste

#### Pasting code in Python is troublesome

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

```
%paste
In [20]: %paste
  def donothing(x):
    return x
## -- End pasted text --
```

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

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

## 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 \mus \pm 1.84 \mus per loop
(mean ± std. dev. of 7 runs, 10000 loops each)
```

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

% %timeit: Several lines



## 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



## 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 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}



#### 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



## Understanding Data Types in Python (I)

```
Static typing

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

- 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
```

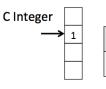
- Data types are not declared
- Data types can change
- Error detection in run-time
- Variables are complex data structures (even for simple types)



## 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];
};
```



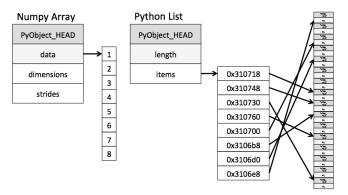




## Understanding Data Types in Python (III)

#### 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]
```





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



#### 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 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)

<b>ДАТА ТҮРЕ</b>	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 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)



## Accessing single elements

#### Unidimensional array

• array[index]

Unidimensional array from the end

• array[-index]

Multidimensional array

• array[row,column]

#### Warning

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



## 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]
```



## 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

#### eneral reshaning

#### 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]
```



#### Concatenation of arrays

#### Three methods to join arrays

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

# np.concatenate() In [ɪ]: x = np.array([ɪ, 2, 3]) In [2]: y = np.array([3, 2, 1]) In [3]: np.concatenate([x, y]) Out [ɪ]: array([ɪ, 2, 3, 3, 2, 1])



## 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]
```

#### nn vstack()



## 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

```
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
```

#### 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



#### 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., $\frac{1}{2} = 1.5$ )
//	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 (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)
```



#### 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

```
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]])
```



## 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

```
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))
```

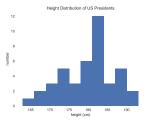


# Universal functions: Aggregations (III)

```
Basic data analysis example (Continuation)

%matplotlib in line
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');
```



# 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

- r. Rule r: 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  $\tau$  in that dimension is stretched to match the other shape.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

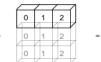


# Universal functions: Broadcasting (II)

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!

# Universal functions: Broadcasting (III)

#### Normalization

```
X = np.random.random((10, 3))

Xmean = X.mean(0)

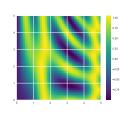
X_centered = X - Xmean
```

### 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)**io+np.cos(io+y*x)*np.cos(x)

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



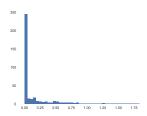
## Comparisons, masks, and Boolean logic (I)

#### (Download dataset)

```
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);
```



# 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



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

```
Syntax examples

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

NumPy

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
1	np.bitwise_or
^	np.bitwise_xor
~	np.bitwise_not



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

```
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 < o) # Returns False
np.all(x < 10)# Returns True
np.sum(~((inches <= 5) | (inches >= 1)))
```

## 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(roo, size=ro)
[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



# 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 (II)

```
Structured array manipulation

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

# Get all names
data ['name']
# Get first row of data
data [o]
# Get the name from the last row
data [-r]['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



#### 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



#### 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 = PANel DAta System
- Built on NumPy's ndarray
- Provides dataframes

#### Pandas provides two main objects

• Series and DataFrame









#### Convention

import numpy as np import pandas as pd



# 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'	0.00



# The Pandas Series object (II)

```
In[1]: data = pd. Series ([0.25, 0.5, 0.75, 1.0],
                    index = ['a', 'b', 'c', 'd'])
In [2]: data
Out [ 1 ]:
  0.25
b 0.50
   0.75
    1.00
dtype: float64
In [3]: data['a']
Out [2]: 0.25
In [4]: data[0]
Out [3]: 0.25
```

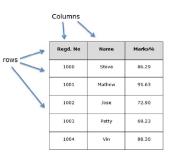
# The Pandas DataFrame object (I)

#### 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



(Source)



# The Pandas DataFrame object (II)

```
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.I
                         3 - 5
                                       I.4
                                                    0.2
                                                         setosa
                                                    0.2
                                                         setosa
            4.9
                         3.0
                                       I.4
                                             o.2 setosa
           4 - 7
                        3.2
                                       1.3
           4.6
                                                o.2 setosa
                        3.I
                                       1.5
           5.0
                        3.6
                                       I.4
                                                0.2
                                                         setosa
In [246]: iris.columns
Out [246]:
Index(['sepal_length', 'sepal_width', 'petal_length',
        'petal_width', 'species'], dtype='object')
```

# 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')



# 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', o.25), ('b', o.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
```

## Data indexing and selection: DataFrame

## Dictionary-like syntax

### Array-like syntax

### Remember indexing conventions

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



# 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
```



# Operating on data (I)

# Pandas fully supports NumPy's

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 (o,
    10, (3,4)))
>>> df = pd. DataFrame (rng. randint (o,
    10, (3,4)), columns = ['A', 'B', 'C'
    , 'D'])
>>> print(df)
>>> np.sin(df * np.pi / 4)
o -7.07e-oi i.o -0.7 i.22e-i6
  7.07e-oi -0.7 -0.7 7.07e-oi
2 I.22e-16 O.O O.7 -7.07e-01
```



# 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
   NaN
  5.0
  9.0
    NaN
dtype: float64
>>> A.add(B, fill_value=0)
   2.0
  5.0
 9.0
    5.0
dtype: float64
```



# 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)



# 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()]
     hello
dtype: object
>>> data.dropna()
     hello
dtype: object
>>> data.fillna(o)
     hello
dtype: object
```



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

Many times we need to combine two or more datasets

• Pandas provides pd.concat(), append() and pd.merge()

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)



Combining datasets: pd.concat()(II)

```
\Rightarrow dfr = pd. DataFrame ([{ 'A': 'Ao', 'B': 'Bo'}, { 'A': 'Ar', 'B': 'Br'
    }])
   df2 = pd. DataFrame([{ 'A': 'A2', 'B': 'B2'}, { 'A': 'A3', 'B': 'B3'
    }])
>> print(df1), print(df2); print(pd.concat([df1, df2]))
   A B
              A B
           o A2
                       B_2
                            o Ao
                                     Во
   Αт
      Ві
           1 A3 B3
                                Αг
                                    Ві
                                A<sub>2</sub> B<sub>2</sub>
                                A<sub>3</sub> B<sub>3</sub>
>> pd.concat([df1, df2], axis=1)
       В
            Α
   Ao
      Bo
           A<sub>2</sub>
   AI BI A3 B3
   dfi.append(df2)
```



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



# Combining datasets: pd.merge() (II)

# One-to-one

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

### Many-to-one

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

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

```
>>> print(dfi); print(df5)
                                                        skills
  employee
                    group
                                         group
        Bob
              Accounting
                                    Accounting
                                                          math
0
                               0
      Take
             Engineering
                                    Accounting
                                                  spreadsheets
      Lisa
             Engineering
                                   Engineering
                                                        coding
                               2
                                                         linux
        Sue
                       HR
                                   Engineering
                                             HR
                                                  spreadsheets
                               4
                                             HR
                                                  organization
>>> pd.merge(df1, df5)
                                    skills
   employee
                     group
        Bob
                                     math
              Accounting
o
       Bob
              Accounting
                            spreadsheets
I
      Jake
                                   coding
             Engineering
      Jake
             Engineering
                                    linux
             Engineering
                                   coding
      Lisa
      Lisa
             Engineering
                                    linux
        Sue
                       HR
                            spreadsheets
        Sue
                       HR
                            organization
```



# 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)



# Combining datasets: pd.merge()(V)

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

# 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
<pre>first(), last()</pre>	First and last item
<pre>mean(), median()</pre>	Mean and median
min(), max()	Minimum and maximun
std(), var()	Standard dev. and variance
mad()	Mean absolute deviation
<pre>prod()</pre>	Product of all items
sum()	Sum of all items
<pre>describe()</pre>	Data summary



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

dtype: float64

# 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)

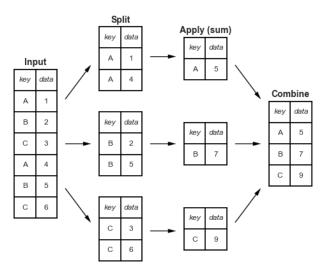
• Example: df . groupby ("key")

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



# Grouping in Pandas (II)





# Grouping in Pandas (III)

```
>>> df = pd. DataFrame ({ 'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                        'data': range(6)})
>>> print(df)
  key data
>>> df.groupby('key')
<pandas.core.groupby.groupby.DataFrameGroupBy object at o</pre>
    x102685438 >
>>> df.groupby('key').sum()
     data
key
```

# 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])



# 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'))



# 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())



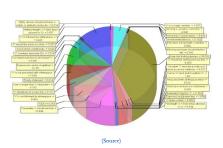
# Grouping in Pandas (VI)

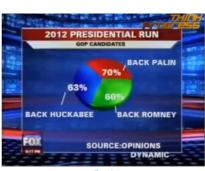
## Grouping by decade

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



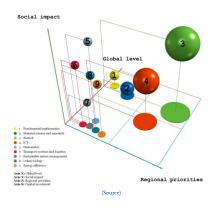
# Bad visualization examples (I)

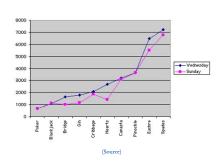






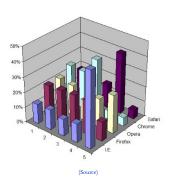
# Bad visualization examples (II)







# Bad visualization examples (III)

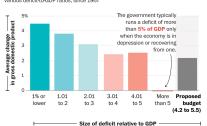




# Bad visualization examples (IV)

#### Strange time for a stimulus

What annual economic growth averaged under various deficit-to-GDP ratios, since 1967

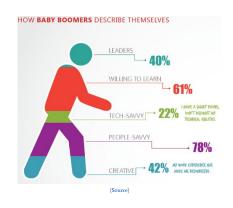


Notes: To capture the environment in which the budget was set, deficit to GDP ratios are compared with the economic climate of the prior fiscal year, GDP growth is adjusted for inflation and seasonality, indicators for the current budget are based on the average of available data in fiscal 2017 and 2018 years. Fiscal years end in September.

Sources: Commerce Department (GDP); Congressional Budget Office (historical deficit); Committee for a Responsible Federal Budget (deficit forecasts, budget changes)

THE WASHINGTON POST

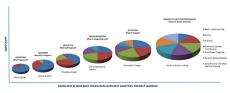
(Source)





#### Visualization

# Bad visualization examples (V)



(Source)



(Source)



#### 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!



#### 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

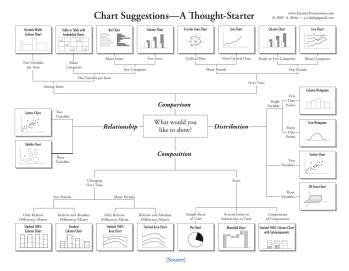
#### What do you want to represent?

• Distribution, relationship, comparison or composition

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



#### Motivation (III)



## 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
- IPython shell.

  Must use %matplotlib
- IPython notebool. Two modes
  - %matplotlib inline
  - %matplotlib notebook

#### Convention

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

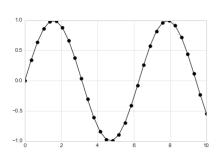
#### myplot.py

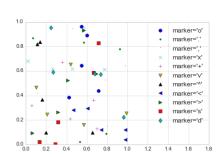
```
import matplotlib.pyplot as plt
import numpy as np
```

```
x = np.linspace(o, 10, 100)
```



# Introduction to Matplotlib (II)





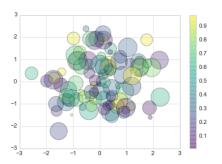
```
for marker in ['o', '.', '.', 'x', '+', 'v', 'A', '
<', '5', 's', 'd'];
plt.plot(rng.rand(5), rng.rand(5), marker,
label="marker="{6}'*.format(marker))
plt.legend(numpoints=r)
plt.xlim(o, r.8);
```



Overview Basics NumPy Pandas Visualization

#### Visualization

# Introduction to Matplotlib (III)



```
rng = np.random.RandomState(o)
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
```

```
0.45

0.40

0.35

0.30

0.25

0.20

0.15

0.10

0.05
```

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

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

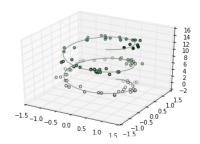


# Introduction to Matplotlib (IV)

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

# Data for a three—dimensional line
zline = np.linspace(o, 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.scatteryD(xdata, ydata, zdata, c=zdata, cmap='
Greens');
```



# 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)



This documentation is for Seaborn 0.9 or newer



## 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



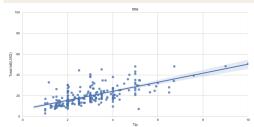
Overview Basics NumPy Pandas Visualization

#### Visualization

## Introduction to Seaborn (III)

## Typical Seaborn usage

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



#### Seaborn datasets (I)

Seaborn comes with several dummy datasets

- Pretty handy
- sns.load\_dataset('name')

We will use two datasets

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

#### Tips datase

 $>>> tips = sns.load\_dataset('tips')$ 

>>> print(tips.head())

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
I	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



## Seaborn datasets (II)

#### Trie datas

>>> tips = sns.load\_dataset('tips')

>>> print(tips.head())

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
I	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 Setosa

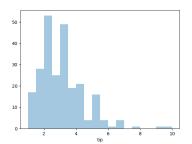


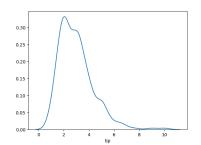
Iris Virginica

(Source)



# Seaborn: Distributions (I)

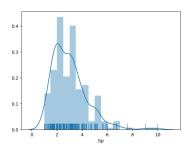




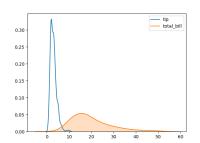
```
sns.distplot(tips['tip'],
    hist = False)
```



# Seaborn: Distributions (II)



#### Histogram , density plat

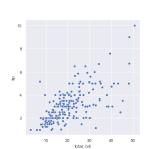


#### Density plot

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



# Seaborn: Relationships (I)



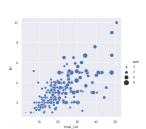
sns.relplot(x="total\_bill", y="
tip", data=tips)

#### n 1

#### Scatterplots



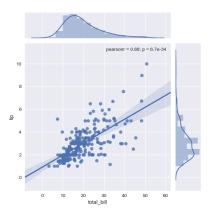
sns.relplot(x="total\_bill", y="
tip", hue="smoker", style="
smoker", data=tips)

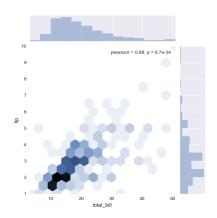






# Seaborn: Relationships (II)



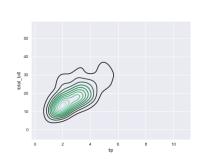


sns.jointplot("total\_bill", "tip", tips, kind="reg"

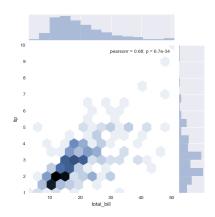
sns.jointplot("total\_bill", "tip", tips , kind="hex



# Seaborn: Relationships (III)



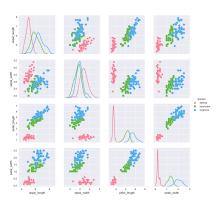
sns.kdeplot(tips['tip'], tips['total\_bill'])



sns.jointplot("total\_bill", "tip", tips , kind="hex

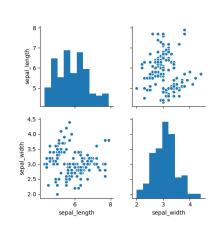


# Seaborn: Relationships (IV)



#### Scatterplot matrix

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

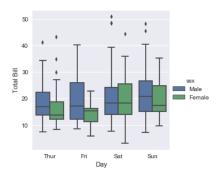


#### Scatternlot matrix

sns.pairplot(iris , vars =[ "sepal\_length" , "
sepal\_width "])

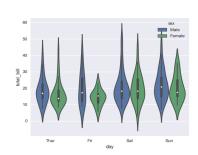


# Seaborn: Comparisons (I)





with sns.axes\_style(style='ticks'):
g = sns.factorplot('day', 'total\_bill', 'sex',
data=tips, kind='box')
g.set\_axis\_labels('Day', 'Total\_Bill')



#### Ziolin plot



# Seaborn: Comparisons (II)

