Python for Data Science

CONSULTING BSTORM

Table of Content

IPython

Beyond Normal Python

CONSULTING BSTORM

Introduction & Installation

IPython is about using Python effectively for interactive scientific and data-intensive computing.

1. Install Anaconda (https://www.anaconda.com/download)

2. In command line:
   a. conda update conda
   b. conda update ipython
   c. ipython

IPython Help

- **help()**
  - allows to access quickly IPython information
- The **?** character allows to explore documentation (buit-in functions, methods, objects, custom functions)
  - ```
    def square(a):
    ....:     """Return the square of a."""
    ....:     return a ** 2
    ```
- The **??** characters allow to explore source code (if no source, it means that the code has been compiled in C (or another language), not in Python)
- The **Tab key** provides auto-completion
  - Functions, objects, ...
- ***Warning?** ⇒ if you know middle characters of a word
    ```

IPython Shell Shortcuts

Keystroke	Action
`Ctrl-a`	Move cursor to the beginning of the line
`Ctrl-e`	Move cursor to the end of the line
`Ctrl-b` or the left arrow key	Move cursor back one character
`Ctrl-f` or the right arrow key	Move cursor forward one character

IPython Shell Shortcuts

Keystroke	Action
Backspace key	Delete previous character in line
`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 (i.e. paste) text that was previously cut
`Ctrl-t`	Transpose (i.e., switch) previous two characters

IPython Shell Shortcuts

Keystroke	Action
`Ctrl-p` (or the up arrow key)	Access previous command in history
`Ctrl-n` (or the down arrow key)	Access next command in history
`Ctrl-r`	Reverse-search through command history

IPython Shell Shortcuts

Keystroke	Action
`Ctrl-l`	Clear terminal screen
`Ctrl-c`	Interrupt current Python command
`Ctrl-d`	Exit IPython session

In & Out Objects

print(In) / print(Out)

In[3]

_ (for accessing the previous output)

_4 ⇒ Out[4]

Shell Command

!shell_command

Examples:

!dir

!echo “Test”

ls = !dir

IPython Magic Commands

Line magics, which are denoted by a single **%** prefix

Cell magics, which are denoted by a double **%%** prefix

```
%run <fileName>
```

```
%timeit or %%timeit
```

```
%magic
```

```
%lsmagic
```

Jupyter Notebook

The Jupyter notebook is a (**browser-based**) **graphical interface to the IPython shell**, and builds on it a rich set of dynamic display capabilities.

As well as executing Python/IPython statements, the notebook allows the user to include formatted text, static and dynamic visualisations, mathematical equations, JavaScript widgets, and much more.

```
$ jupyter notebook
```

This command will launch a local web server that will be visible to your browser.

NumPy

Introduction

NumPy provides techniques for effectively loading, storing, and manipulating in-memory data in Python.

Datasets can come in a **wide range of formats** (collections of documents, images, sound clips, numerical measurements, …)

Despite this apparent heterogeneity, it will help us to think of all data fundamentally as **arrays of numbers**. (Example: images are two-dimensional arrays of numbers representing pixel).

NumPy (short for Numerical Python) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python's built-in list type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size.

NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

Basic Instructions

```
import numpy
numpy.__version__
import numpy as np

np.<TAB>

np?
```

Data Types in Python

How NumPy improves the way Python handles data?

Python is dynamic typing (in contrast with a statically-typed language like C or Java)

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

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

```c
/* C code */
int x = 4;
x = "four";  // FAILS
```

```python
# Python code
x = 4
x = "four"
```

Data Types in Python

It implies that Python variables are more than just their value

They also contain **extra information** about the type of the value

What Is a Python Integer?

Python implementation is written in C → Every Python object is simply a cleverly-disguised C structure which contains not only its value, but other information as well.

For example, when we define an integer in Python, such as x = 10_000, x is not just a "raw" integer. It's actually a pointer to a compound C structure, which contains several values. Looking through the Python 3.x source code, we find that the integer (long) type definition effectively looks like this (once the C macros are expanded):

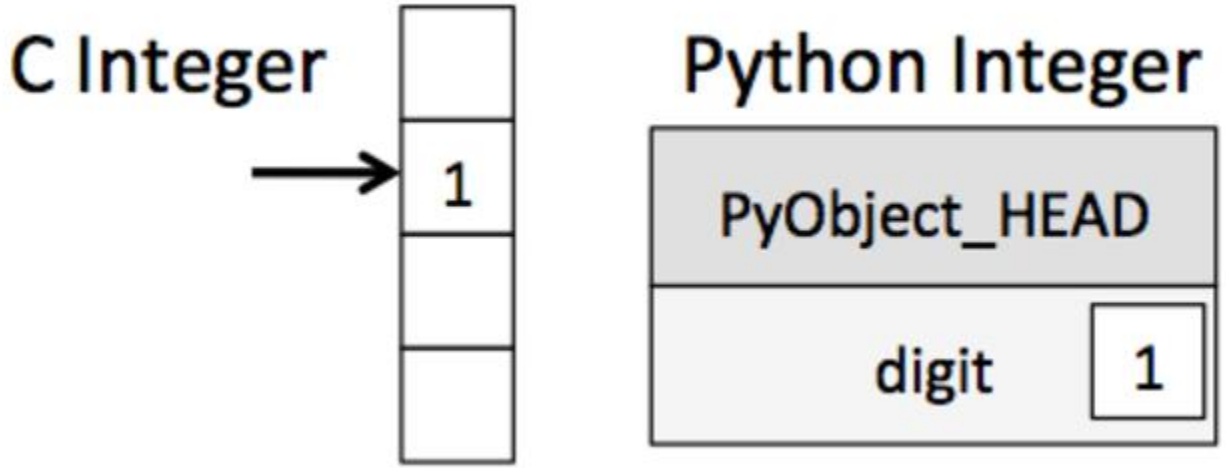
A single integer in Python 3.x actually contains four pieces:

- ob_refcnt, a reference count that helps Python silently handle memory allocation and deallocation
- ob_type, which encodes the type of the variable
- ob_size, which specifies the size of the following data members
- ob_digit, which contains the actual integer value that we expect the Python variable to represent

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

What Is a Python Integer?

This means that there is some overhead in storing an integer in Python as compared to an integer in a compiled language like C, as illustrated in the following figure:



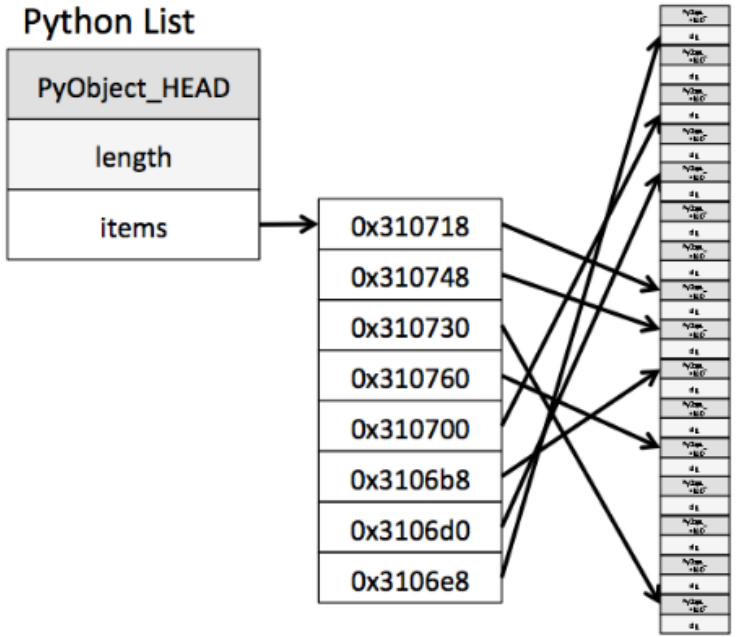
What Is a Python List?

What happens in a Python data structure that holds many Python objects?

```
L = list(range(10))
type(L[0])
L2 = [str(c) for c in L]
```

We can even create heterogeneous lists:

```
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
```

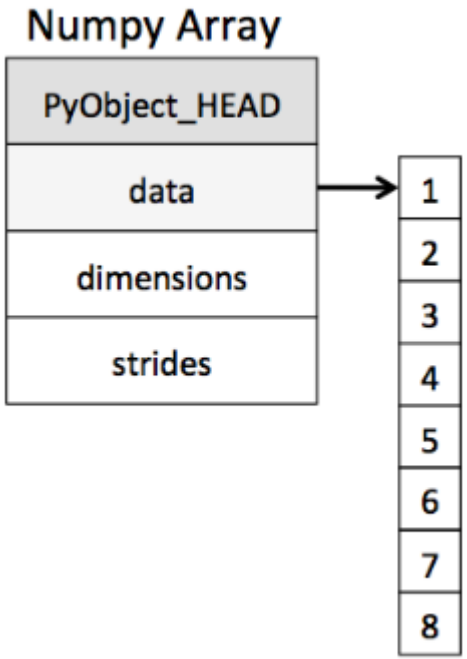


What Is a Numpy Array?

In the special case that all variables are of the same type, much of this information is **redundant**.

The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in the following figure.

Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.



Create a NumPy Array

```
np.array([1, 4, 2, 5, 3])
```

NumPy is constrained to arrays that **all contain the same type**. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

```
np.array([3.14, 4, 2, 3])
```

If we want to explicitly **set the data type** of the resulting array, we can use the dtype keyword:

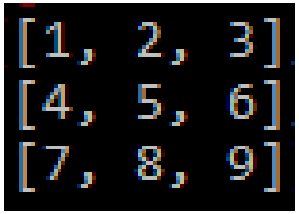
```
np.array([1, 2, 3, 4], dtype='float32')
```

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensional; here's one way of initialising a multidimensional array using a list of lists:

```
# nested lists result in multi-dimensional arrays
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

Exercise

1. Create the following 3x3 matrix



Create NumPy Array from Scratch

```
# Create a length-10 integer array filled with zeros
np.zeros(10, dtype=int)

# Create a 3x5 floating-point array filled with ones
np.ones((3, 5), dtype=float)

# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)

# Create an array filled with a linear sequence starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)

# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)

# Create a 3x3 array of uniformly distributed
# random values between 0 and 1
np.random.random((3, 3))  // How can you display all distributions available in NumPy?

# Create a 3x3 array of normally distributed random values with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
```

Standard Data Types in NumPy

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

NumPy Array Attributes

```
np.random.seed(0)  # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array

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

Array Indexing: Accessing Single Elements

Positive indexing:

$$\texttt{array\_name[i]} \quad \Rightarrow \quad 0 \leq i < \texttt{array\_name.shape[0]}$$

Negative indexing:

$$\texttt{array\_name[-i]} \quad \Rightarrow \quad \texttt{-array\_name.shape[0]} \leq i \leq -1$$

Two-dimensions array:

```
x2[0, 0]
```

Array Slicing

```
x[start:stop:step]      # select from start to stop (not included) by step
```

Default value:

start = 0

stop = size

step = 1

(if step is negative, default values for *start* and *stop* are reversed)

Array Slicing – 1 Dimension

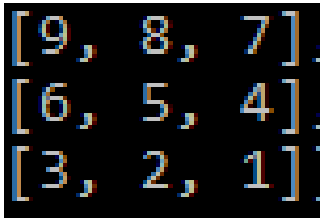
Exercises

1. Create a table with integers ranging from 0 to 9
2. Extract the 5 first elements
3. Extract the elements following the 5th one
4. Extract elements 4 to 7
5. Extract even elements
6. Extract uneven elements (without 0)
7. Extract the last 3 elements
8. Extract elements in a reversed order
9. Extract the 5 first elements, in a reversed order
10. Select all elements using the slicing syntax

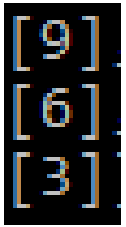
Array Slicing – 2 Dimensions

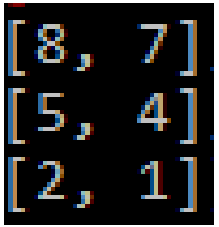
Exercises

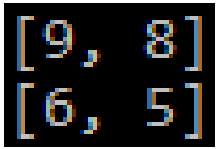
1. Create the following matrix

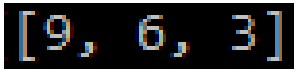


2. Print the following sub-matrices

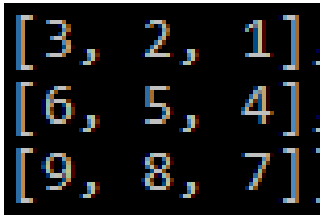








3. Invert the lines



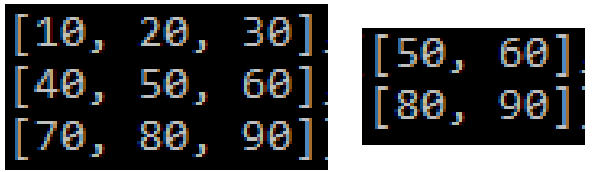
Views

One important – and extremely useful – thing to know about array slices is that they return *views* rather than *copies* of the array data.

This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:

Exercises:

1. Create the following matrix in x1 variable:
2. Extract it in x2 variable
3. Replace “50” in x2 by “99”
4. Print x1 again



Copy

Use the copy() method on a NumPy array to

Exercises:

1. Create the following matrix in x1 variable:
2. Extract a copy of it in x2 variable
3. Replace “50” in x2 by “99”
4. Print x1 again

```
[10, 20, 30]
[40, 50, 60]
[70, 80, 90]
```

```
[50, 60]
[80, 90]
```

Reshape Array

`ndarray.reshape(tuple) # no-copy`

Example

```
np.arange(1, 10).reshape((3, 3))
```

Transpose

x.T

Transform a Vector Into a Column/Line Matrix

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

# row vector via reshape
x.reshape((1, 3))

# row vector via newaxis
x[np.newaxis, :]
x.reshape((3, 1))

# column vector via newaxis
x[:, np.newaxis]
```

np.concatenate

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

RESULT

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

```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])

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

# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

np.Xstack

np.hstack

```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

np.vstack

```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])

# vertically stack the arrays
np.vstack([x, grid])
```

np.dstack (third dimension (deepness))

np.split

SPLIT:

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

RESULT

```
[1 2 3] [99 99] [3 2 1]
```

VSPLIT:

```
grid = np.arange(16).reshape((4, 4))
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

RESULT

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

HSPLIT:

```
left, right =
np.hsplit(grid, [2])
print(left)
print(right)
```

RESULT

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Computation Time and UFunc

Create a function that compute a new array which is the reciprocal of the input, then time it on this array big_array

```
big_array = np.random.randint(1, 100, size=1000000)
```

NumPy Arithmetic Operators (& Binary 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)

Unary Functions

np.abs()

np.sin(), np.cos(), np.tan()

np.exp(x) # e^x

np.exp2(x) # 2^x

np.log(x) # ln(x)

np.log2(x) # $\log_2(x)$

out Parameter

In order to avoid the creation of a temporary array

```
x = np.arange(5)
np.multiply(x, 10, out=y)
```

or

```
y = np.zeros(10)
np.power(2, x, out=y[::2]) # faster than y[::2] = 2 ** x
print(y)
```

Aggregation of Binary Functions

`np.[Ufunct].reduce([array])` - Aggregates all elements using Ufunct

`np.[Ufunct].accumulate([array])` - Aggregates all elements using Ufunct and returns intermediate results

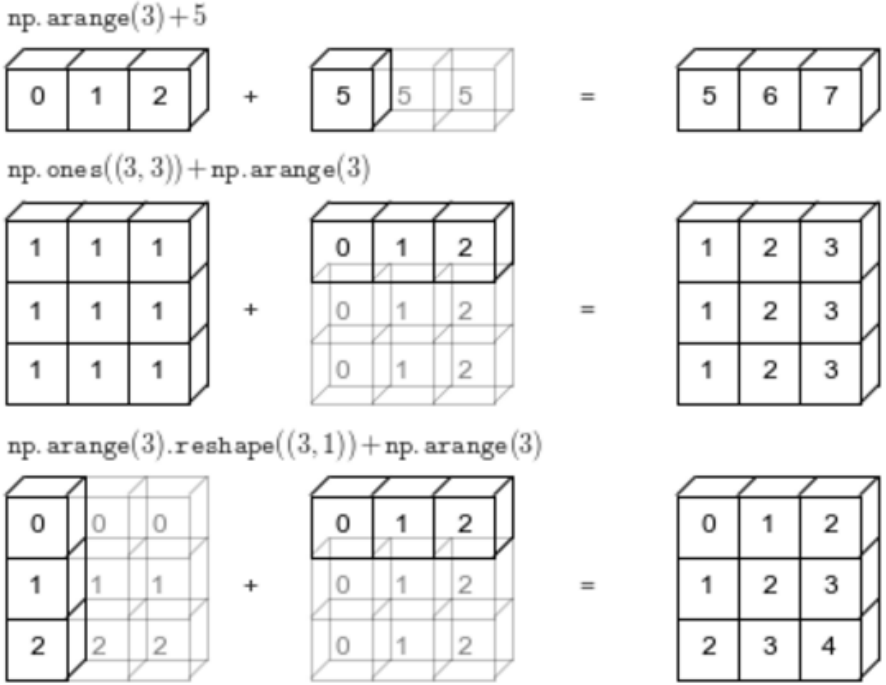
`np.[Ufunct].outer([array1],[array2])` - Apply Ufunct for each pair of elements from the two parameters

Aggregation Functions

The axis index can be used to aggregate in one direction.

Function Name	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 variance
`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

Broadcasting of Binary Functions



Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

- **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.
- **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.
- **Rule 3:** if in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Exercices

Create a function that will calculate the euclidean distance between observations from a table.

Use this dataset as test:

Try to optimise the algorithm in order to be executed under 12 sec

Data Manipulation with Pandas

CONSULTING BSTORM

Import the Package

```
import pandas as pd
```

Pandas Data Structures

Series: is a one-dimensional array of indexed data.

DataFrame: is an analog of a two-dimensional array with both flexible row indices and flexible column names. It is a collection of Series objects.

Index: is an immutable ordered multiset of localisation.

Pandas Series

Building:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
```

Wraps a sequence of indices and a sequence of values (which is a NumPy array):

```
data.index
data.values
```

Because indices are explicit (unlike basic NumPy arrays), they do not need to be integer.

```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
```

There are simultaneously explicit and implicit indices !!!

Pandas DataFrames

```
states = pd.DataFrame({'population': population, 'area': area})

states['colIndex']['rowIndex']
```

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

Building Pandas DataFrames

From a single Series object

```
pd.DataFrame(population, columns=['population'])
```

	population
California	38332521
Florida	19552860
Illinois	12882135
New York	19651127
Texas	26448193

From a list of dicts

```
pd.DataFrame([{'a': i, 'b': 2 * i} for i in range(3)])
```

From a dictionary of Series objects

```
pd.DataFrame({'population': population, 'area': area})
```

From a two-dimensional NumPy array

```
pd.DataFrame(np.random.rand(3, 2), columns=['foo', 'bar'], index=['a', 'b', 'c'])
```

From a CSV file

```
pd.read_csv(PATH)
```

Pandas Index Object

- Index as immutable arrays
  - Standard indexing notation (*i.e.* slicing) can be used
  - Cannot be modified with direct assignation

- Index are ordered (multi-) set
  - ```
    indA = pd.Index([1, 3, 5, 7, 9])
    indB = pd.Index([2, 3, 5, 7, 11])
    ```
  - ```
    indA & indB  # intersection
    ```
  - ```
    indA | indB  # union
    ```
  - ```
    indA ^ indB  # symmetric difference
    ```

Data Indexing & Selection

```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index = ['a', 'b', 'c', 'd'])
data['b'] # 0.5
'a' in data # true
data.keys() # returns the indices objects
list(data.items()) # returns a list consisting of rows as tuple
data['e'] = 1.25 # assign the value to the localisation
data['a':'c'] # slicing by explicit index
data[0:2] # slicing by implicit integer index
data[(data > 0.3) & (data < 0.8)] # select element by masking
data[['a', 'e']] # use of fancy indexing
```

Confusion with Implicit & Explicit Indices

What happens with…?

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data[1] # explicit index when indexing
data[1:3] # implicit index when slicing
```

loc → use of the explicit indices

```
data.loc[1] # 'a'
data.loc[1:3]
```

iloc → use of the implicit indices

```
data.iloc[1]
data.iloc[1:3]
```

Add a New Column

```
data['density'] = data['pop'] / data['area']
```

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

Operating on Data in Pandas

- Any NumPy UFuncs works on Series or DataFrames

- Missing indices results in NaN
  - ```
    A = pd.Series([2, 4, 6], index=[0, 1, 2])
    B = pd.Series([1, 3, 5], index=[1, 2, 3])
    A + B
    ```
  - Otherwise: `A.add(B, fill_value=0)`

Python Operator	Pandas Method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()
    ```

Handling Missing Values

```
df.isnull()

df.notnull()

df.dropna()

df.dropna(axis=1)

df.dropna(axis=1, how=”all”)

df.dropna(axis='rows', thresh=3)
# specify a min. number of non-null values for the row/column to be kept

df.fillna(0)

df.fillna(method='ffill', axis=1) # propagate the previous value forward

df.fillna(method='bfill', axis=0) # propagate the next values backward
```

Combining Datasets: concat & append

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])

pd.concat([ser1, ser2])
pd.concat([ser1, ser2], axis=1)
```
→ If duplicated indices, it duplicates indices

```
pd.concat([x, y], verify_integrity=True)
```
→ If duplicated indices, it raises an exception

```
pd.concat([x, y], ignore_index=True)
```
→ Ignores indices

```
pd.concat([df5, df6], join='inner')
```
→ Consider only matching columns (instead of filling with NaN)

```
pd.concat([df5, df6], join_axes=[df5.columns])
```
→ Consider only columns from df5

Combining Datasets: merge

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
df3 = pd.merge(df1, df2)
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Combining Datasets: merge

```
Specifying the merge key
pd.merge(df1, df2, on='employee')

pd.merge(df1, df3, left_on="employee", right_on="name")
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df3

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df1, df3, left_on="employee", right_on="name")

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

Combining Datasets: merge

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')

pd.merge(df1a, df2a, left_index=True, right_index=True)
```

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

pd.merge(df1a, df2a, left_index=True, right_index=True)

	group	hire_date
employee		
Lisa	Engineering	2004
Bob	Accounting	2008
Jake	Engineering	2012
Sue	HR	2014

Combining Datasets: merge

```
pd.merge(df6, df7, how='outer')

# inner, left, right
```

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7, how='outer')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

Aggregate

.describe()

	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510
std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

Aggregate

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 deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

Aggregate + GroupBy

```
df.groupby('key').sum()
df.groupby('key').aggregate(['min', np.median, max])
```

key must be a column (NOT AN INDEX)

use parameter *level* for grouping by index

Example: df.groupby(level=0).mean()

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 deviation and variance
mad()	Mean absolute deviation
prod()	Product of all items
sum()	Sum of all items

Visualisation with Matplotlib

CONSULTING BSTORM

Import & Configuration

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



plt.style.use('classic')
```

Show a Plot in the Shell

```
import numpy as np

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

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

plt.show()
```

If using the *matplotlib* option in IPython / Jupyter, show() is no more required:

```
%matplotlib
```

Saving the Plot as an Image

```
fig.savefig('my_figure.png')
```

```
fig.canvas.get_supported_filetypes()
```

Matlab-like Interface

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

# 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))
```

Line Plot

Colour

```
plt.plot(x, np.sin(x - 0), color='blue')        # specify color by name

plt.plot(x, np.sin(x - 1), color='g')           # short color code (rgbcmyk)

plt.plot(x, np.sin(x - 2), color='0.75')        # grayscale between 0 and 1

plt.plot(x, np.sin(x - 3), color='#FFDD44')     # Hex code (RRGGBB [00-FF])

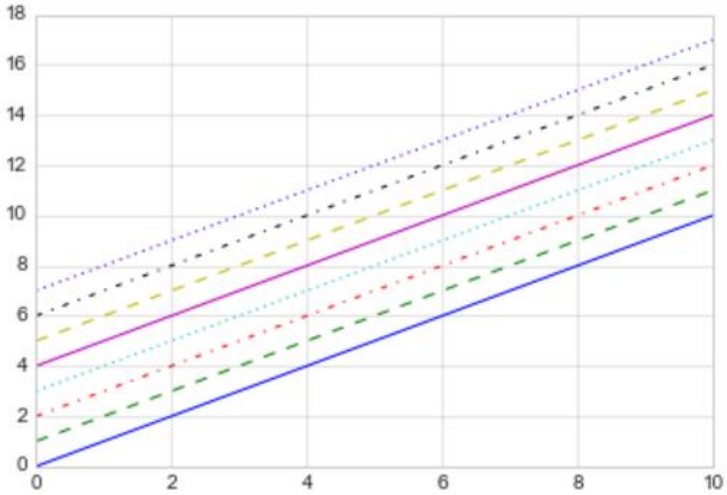
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 to 1

plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML color names supported
```

Line Style

```
plt.plot(x, x + 0, linestyle='solid’)
plt.plot(x, x + 1, linestyle='dashed’)
plt.plot(x, x + 2, linestyle='dashdot’)
plt.plot(x, x + 3, linestyle='dotted');

# For short, you can use the following codes:
plt.plot(x, x + 4, linestyle='-')  # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':');  # dotted
```



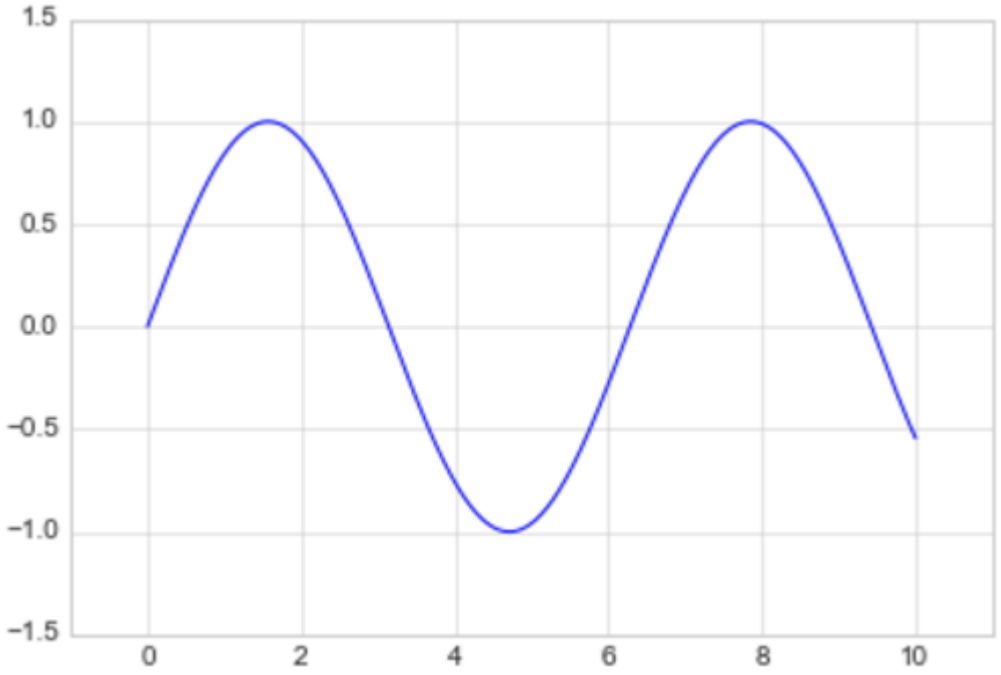
Adjusting the Axis

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

plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5)
```

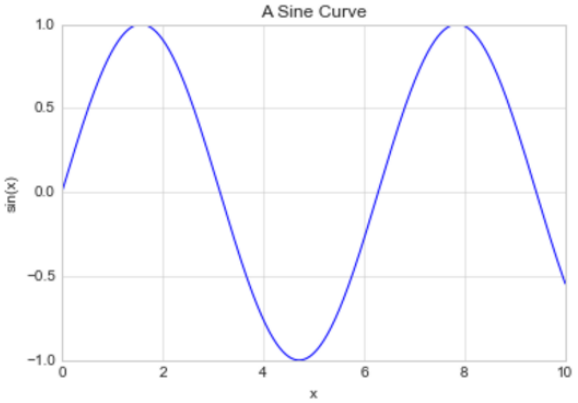
or:

```
plt.axis([-1, 11, -1.5, 1.5])
# [xmin, xmax, ymin, ymax]
```



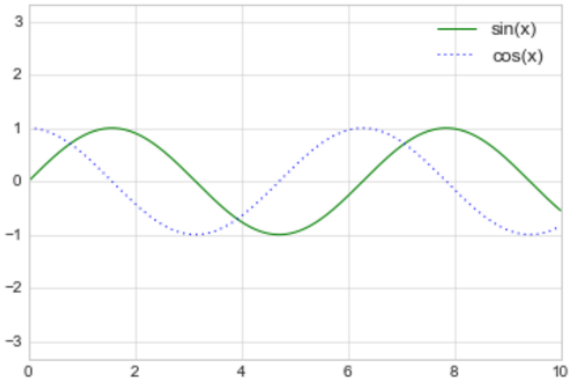
Labeling Plot

```
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```




```
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
plt.axis('equal')

plt.legend();
```

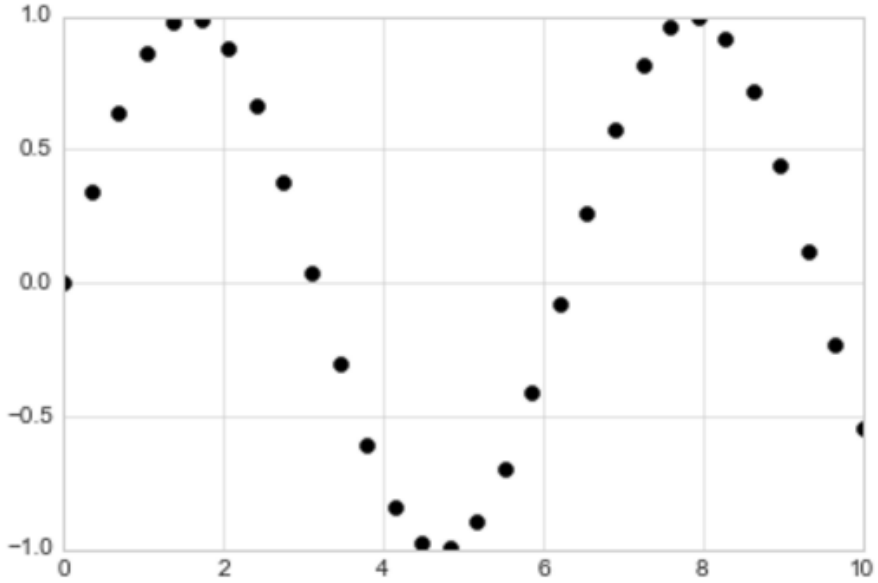


Scatter Plot

Scatter Plot Creation

```
x = np.linspace(0, 10, 30)
y = np.sin(x)

plt.plot(x, y, 'o', color='black')
```

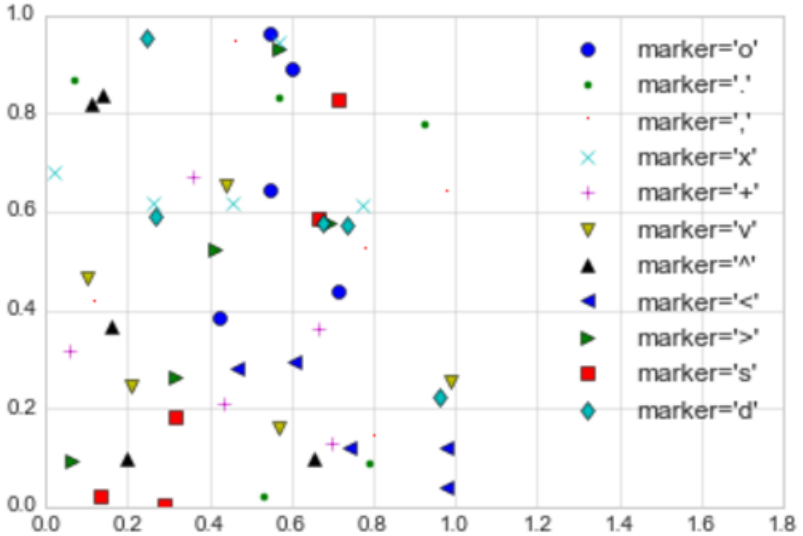


Different Markers

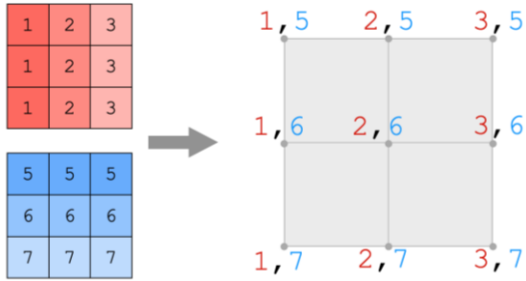
```
plt.plot(rng.rand(5), rng.rand(5), 'o', label="marker='o'")
plt.plot(rng.rand(5), rng.rand(5), '+', label="marker='+'")
```

```
plt.legend(numpoints=1)

plt.xlim(0, 1.8);
```



3D Visualisations



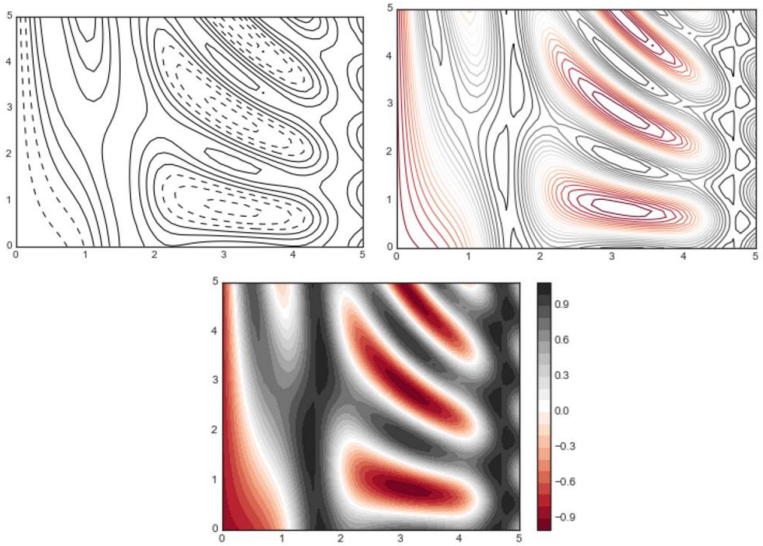
plt.contour

```
def f(x, y): return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

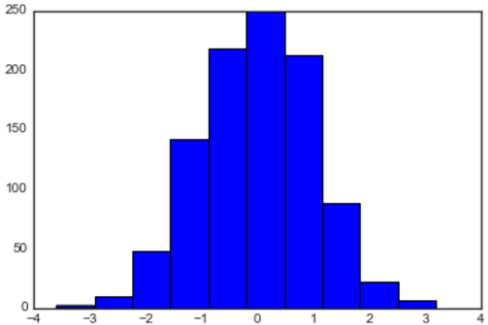

X, Y = np.meshgrid(x, y)
Z = f(X, Y)
plt.contour(X, Y, Z, colors='black');   #1


plt.contour(X, Y, Z, 20, cmap='RdGy');  #2

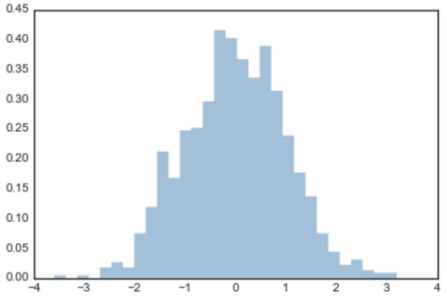
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar();
```



Histograms

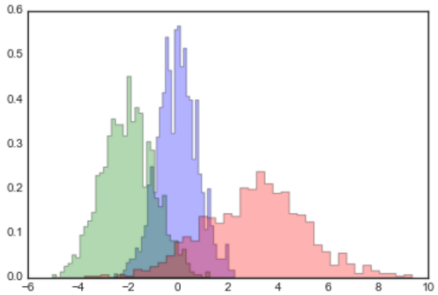



```
data = np.random.randn(1000)
plt.hist(data);
```




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

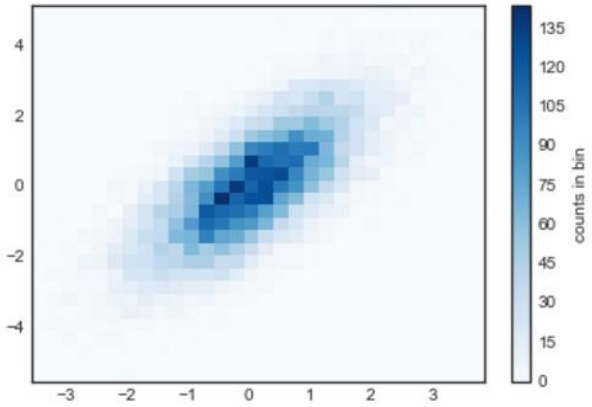
```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)
kwargs = dict(histtype='stepfilled', alpha=0.3,
              normed=True, bins=40)
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs)
```



Two-dimensional Histograms

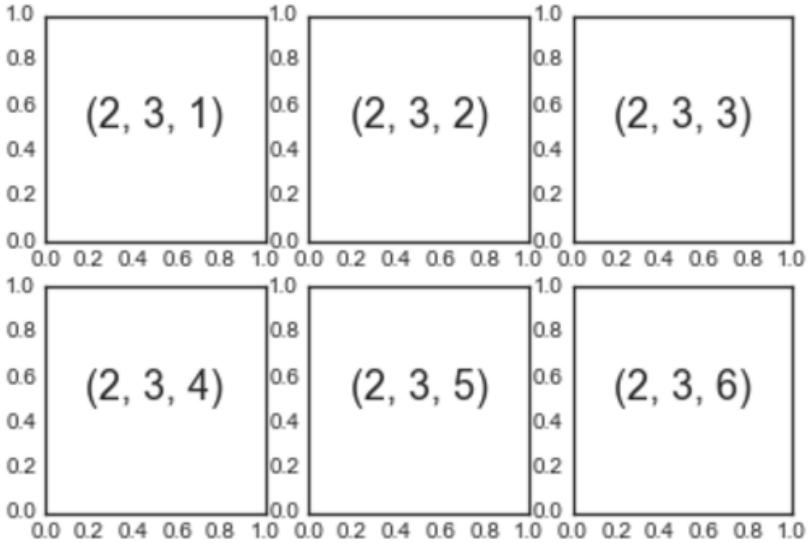
```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

```
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')
```



Subplots

```
for i in range(1, 7):
    plt.subplot(2, 3, i)
    plt.text(0.5, 0.5, str((2, 3, i)),
             fontsize=18, ha='center')
```



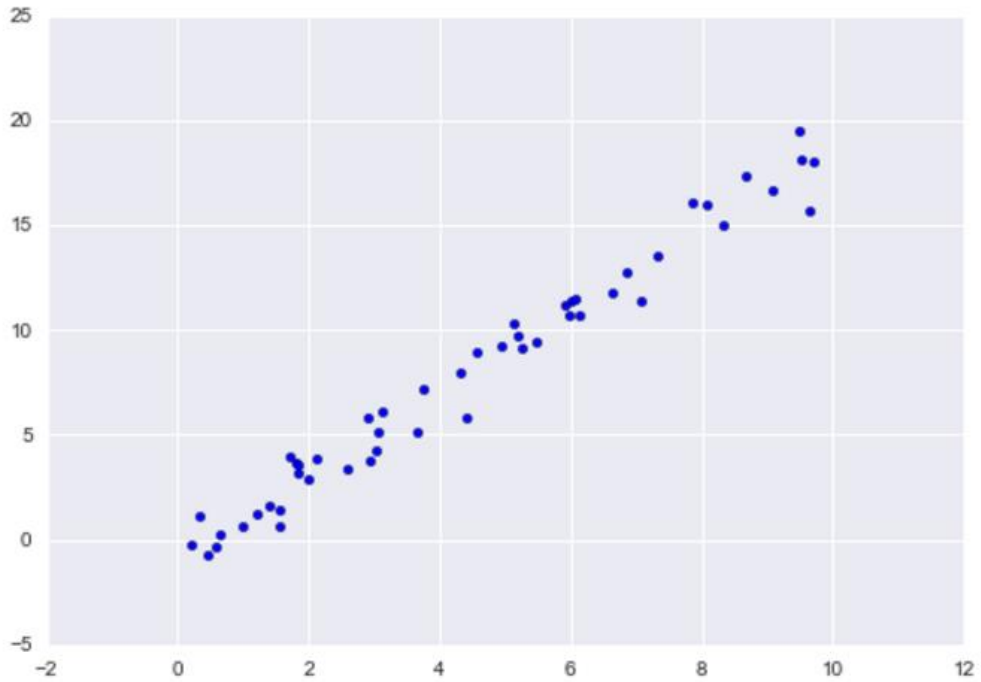
Machine Learning Algorithms

CONSULTING BSTORM

Simple Linear Regression

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

rng = np.random.RandomState(42)
x = 10 * rng.rand(50)
y = 2 * x - 1 + rng.randn(50)
plt.scatter(x, y)
```



Simple Linear Regression

1. **Choose a class of model**

```
from sklearn.linear_model import LinearRegression
```

2. **Choose model hyperparameters**

```
model = LinearRegression(fit_intercept=True)
```

3. **Arrange data into a features matrix and target vector**

```
X = x[:, np.newaxis]
```

4. **Fit the model to your data**

```
model.fit(X, y)
model.coef_
model.intercept_
```

5. **Predict labels for unknown data**

```
xfit = np.linspace(-1, 11)
Xfit = xfit[:, np.newaxis]
yfit = model.predict(Xfit)
plt.scatter(x, y)
plt.plot(xfit, yfit)
```

Naive Bayes Classification

1. **Choose a class of model**

```
from sklearn.naive_bayes import GaussianNB
```

2. **Choose model hyperparameters**

```
model = GaussianNB()
```

3. **Arrange data into a features matrix and target vector**

```
from sklearn.cross_validation import train_test_split
X_iris = iris.drop('species', 1)
y_iris = iris['species']
Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris, random_state=1)
```

4. **Fit the model to your data**

```
model.fit(Xtrain, ytrain)
```

5. **Predict labels for unknown data**

```
y_model = model.predict(Xtest)
```

6. **Evaluate**

```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)
```

Principal Component Analysis (PCA)

1. **Choose a class of model**

```
from sklearn.decomposition import PCA
```

2. **Choose model hyperparameters**

```
model = PCA(n_components=2)
```

3. **Arrange data into a features matrix and target vector**

```
X_iris = iris.drop('species', 1)
```

4. **Fit the model to your data**

```
model.fit(X_iris)
```

5. **Predict labels for unknown data**

6. **Evaluate**

```
X_2D = model.transform(X_iris)
iris['PCA1'] = X_2D[:, 0]
iris['PCA2'] = X_2D[:, 1]
sns.lmplot("PCA1", "PCA2", hue='species', data=iris, fit_reg=False)
```

Clustering – K-Means

1. **Choose a class of model**

```
from sklearn.cluster import KMeans
```

2. **Choose model hyperparameters**

```
model = KMeans(n_clusters=4)
```

3. **Arrange data into a features matrix and target vector**

```
X_iris = iris.drop('species', 1)
```

4. **Fit the model to your data**

```
model.fit(X_iris)
```

5. **“Predict” labels for unknown data**

```
y_kmeans = kmeans.predict(X)
```

6. **Evaluate**

```
plt.scatter(X_iris.iloc[:,0], X_iris.iloc[:,1], c=y_kmeans, s=50, cmap='viridis')
centers = model.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5)
```

Clustering

1. **Choose a class of model**

```
from sklearn.mixture import GMM
```

2. **Choose model hyperparameters**

```
model = GMM(n_components=3, covariance_type='full')
```

3. **Arrange data into a features matrix and target vector**

```
X_iris = iris.drop('species', 1)
```

4. **Fit the model to your data**

```
model.fit(X_iris)
```

5. **Predict labels for unknown data**

6. **Evaluate**

```
iris['cluster'] = y_gmm
sns.lmplot("PCA1", "PCA2", data=iris, hue='species', col='cluster', fit_reg=False)
```

Decision Tree

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier().fit(X, y)
```

Confusion Matrix

```
from sklearn.metrics import confusion_matrix

mat = confusion_matrix(ytest, y_model)

sns.heatmap(mat, square=True, annot=True, cbar=False)
plt.xlabel('predicted value')
plt.ylabel('true value')
```

Exercice

In probability theory, the **central limit theorem** (**CLT**) establishes that, for the most commonly studied scenarios, when independent random variables are added, their sum tends toward a normal distribution (commonly known as a *bell curve*) even if the original variables themselves are not normally distributed.

Avec un lancé de deux dés, montrer avec 100 jets, que la distribution suit une distribution normale.