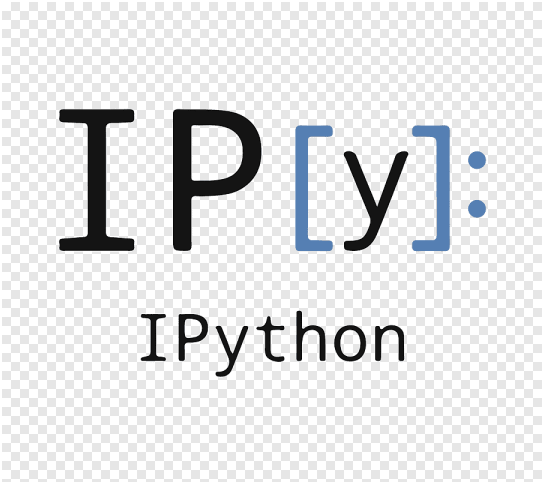


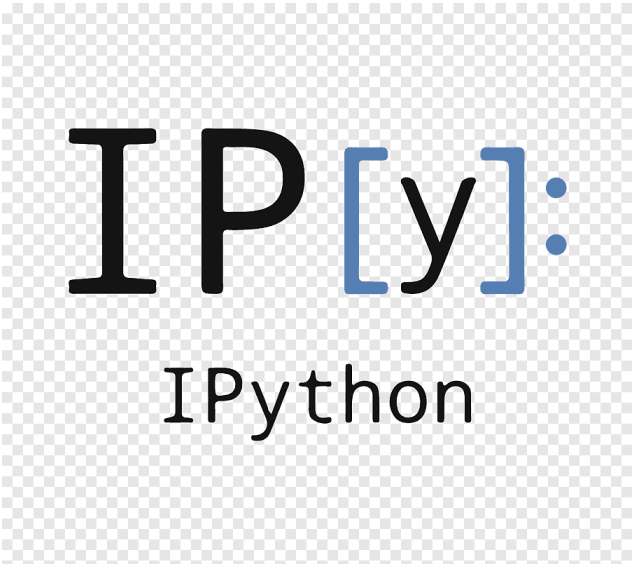

# IPython
# Numpy
# Pandas
# MatPlotLib



Luís Garmendia

# Index 2

# IPython
# Jupiter Notebook





Luís Garmendia

# IPython

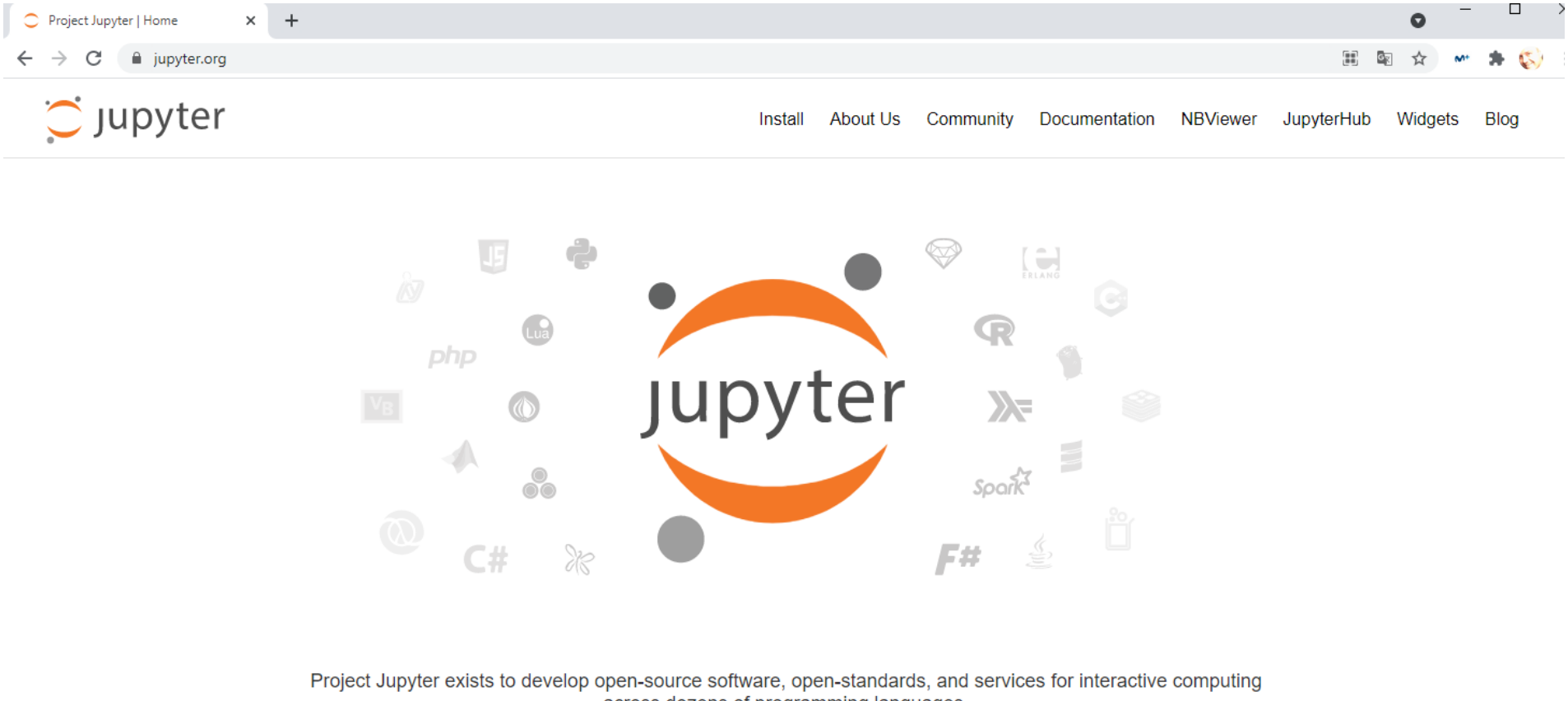
IPython (short for *Interactive Python*) was started in 2001 by Fernando Perez as an enhanced Python interpreter, and has since grown into a project aiming to provide, in Perez's words

"Tools for the entire life cycle of research computing."

If Python is the engine of our data science task, you might think of IPython as the interactive control panel.

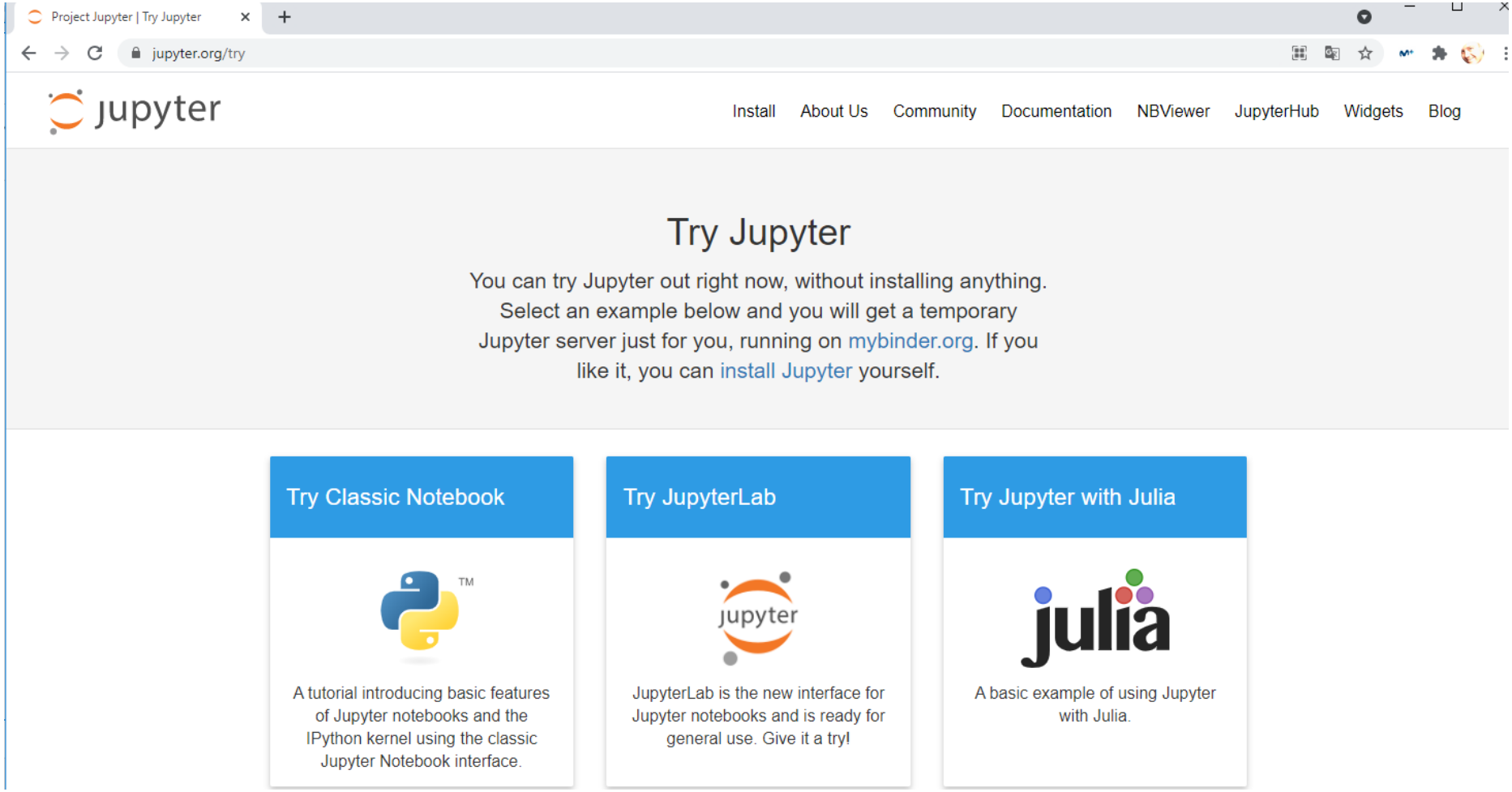
# Jupyper Notebook

https://jupyter.org/

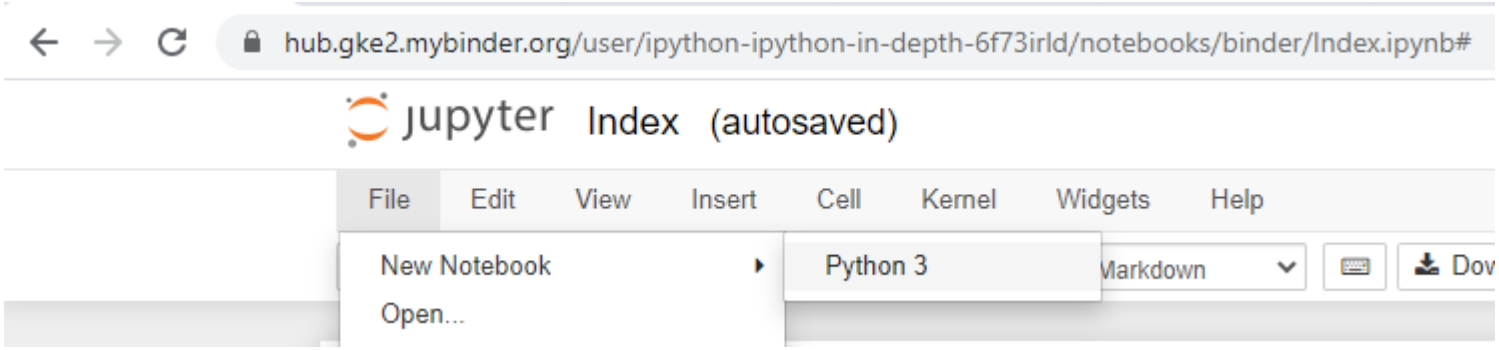


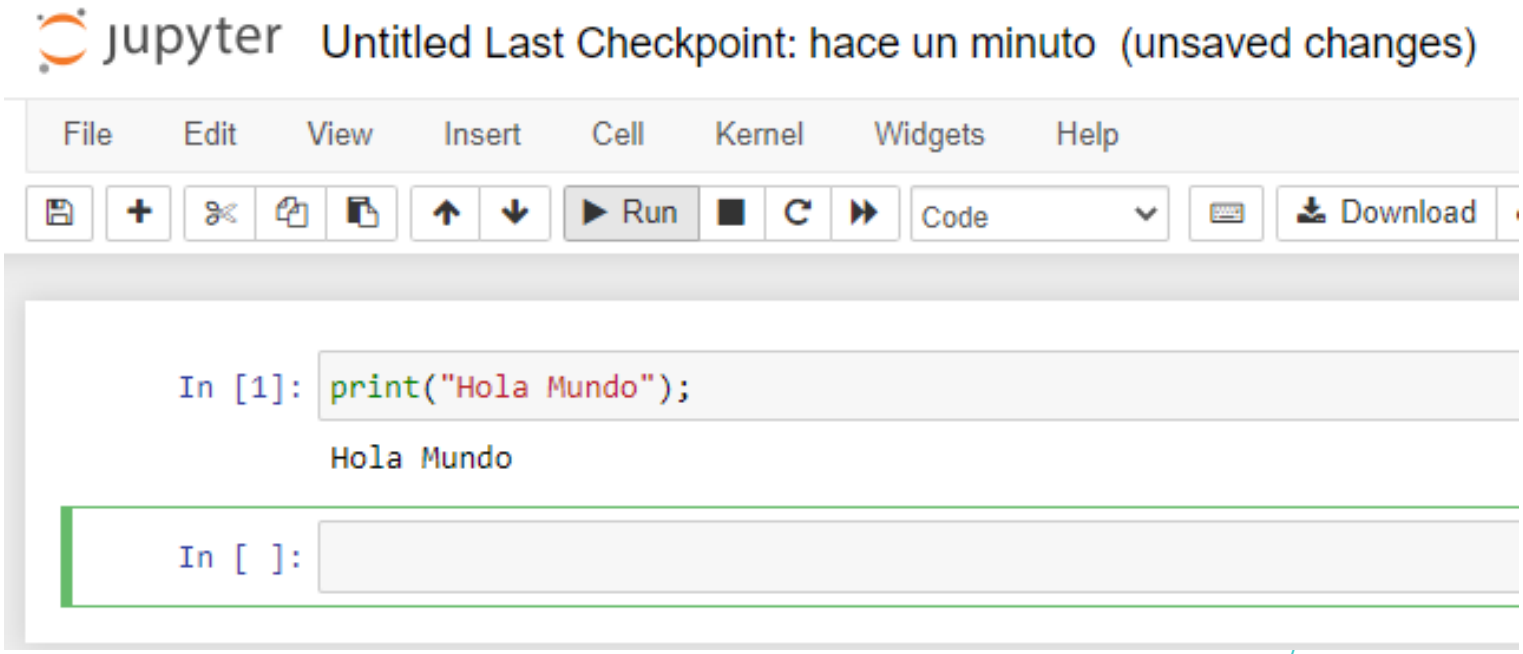
# Jupyper Notebook

https://jupyter.org/try



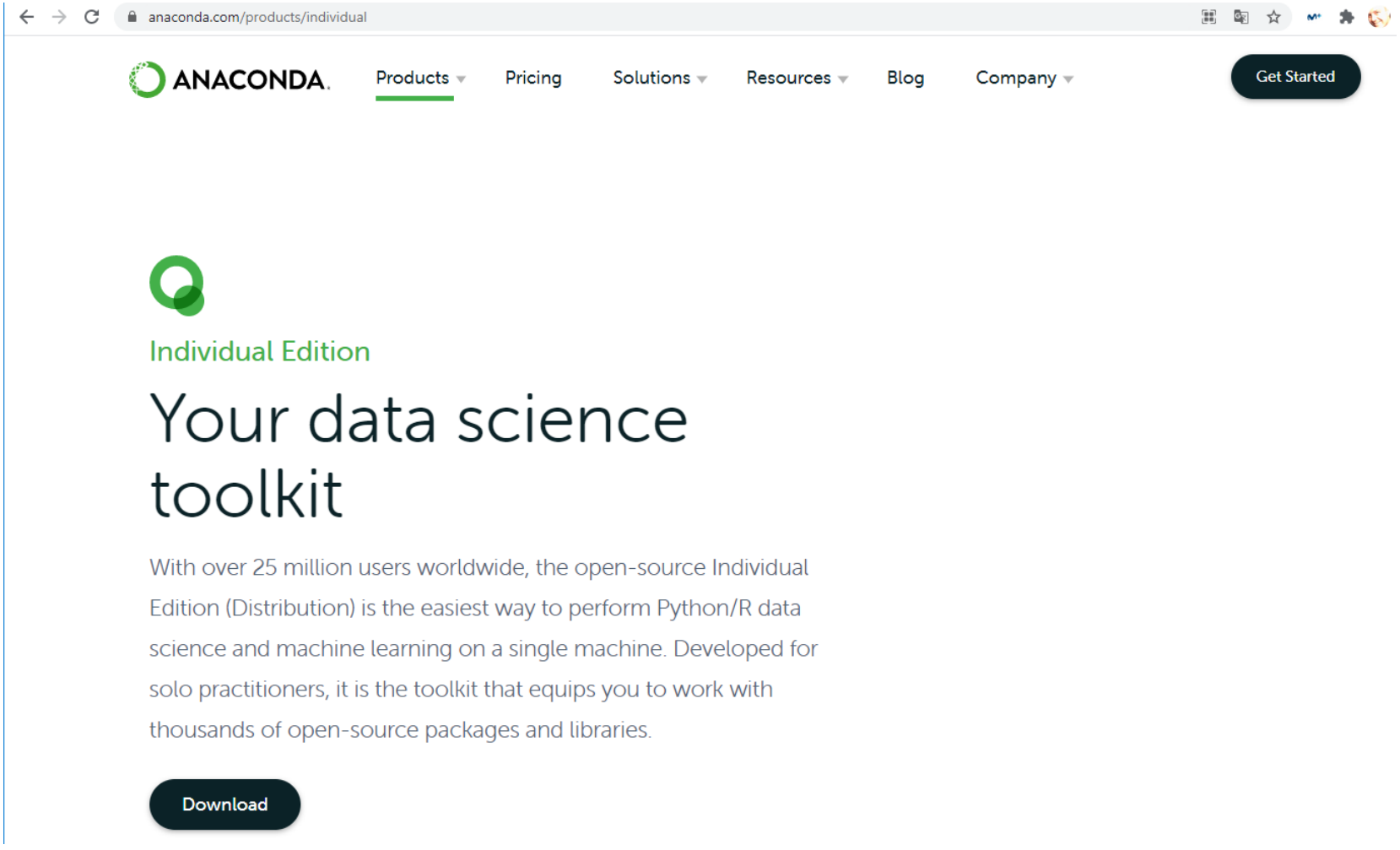
# Jupyper Notebook



# Install Anaconda

https://www.anaconda.com/products/individual



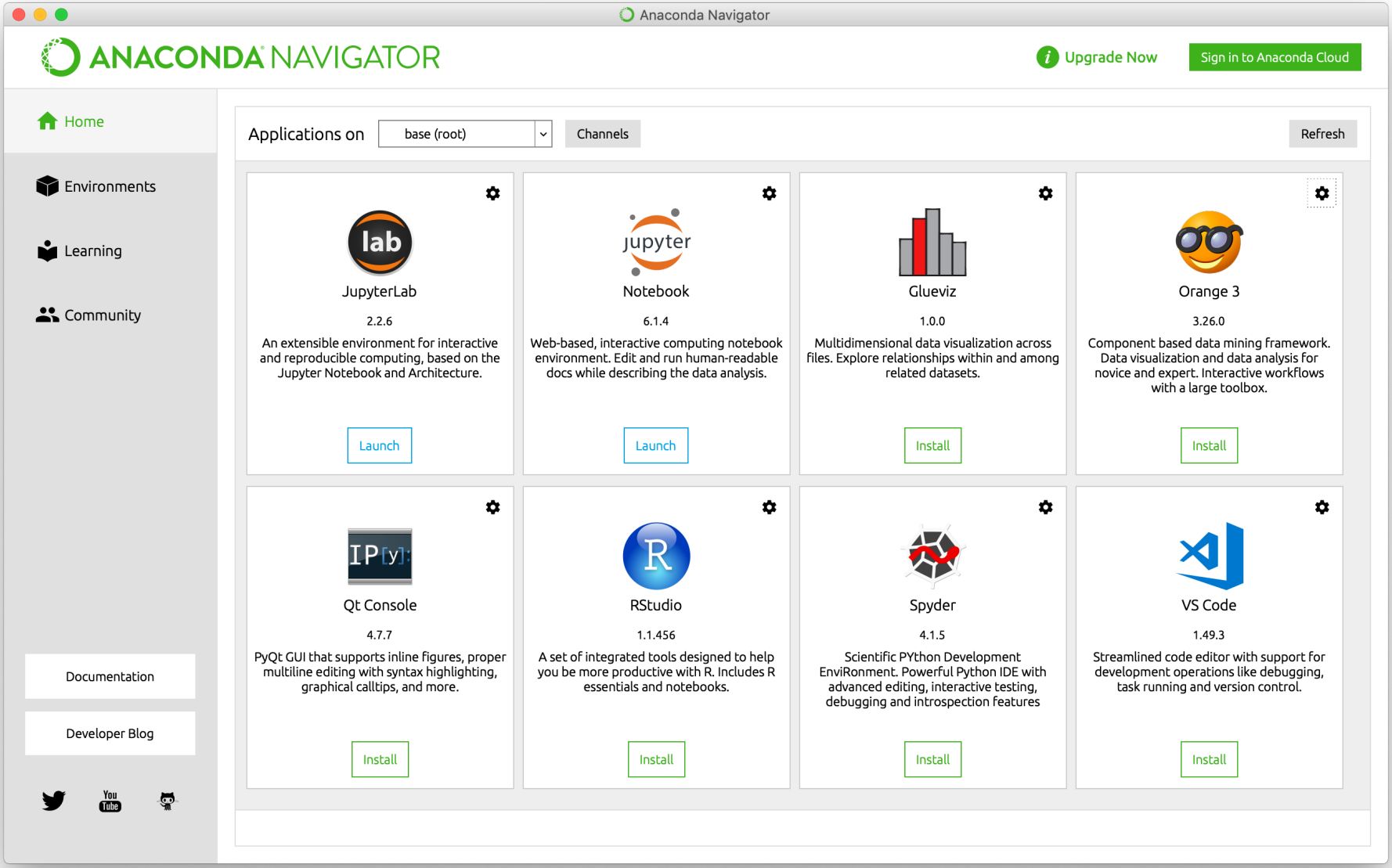
# Anaconda

https://es.wikipedia.org/wiki/Anaconda_(distribuci%C3%B3n_de_Python)

**Anaconda** es una distribución libre y abierta[1] de los lenguajes Python y R, utilizada en ciencia de datos, y aprendizaje automático (*machine learning*). Esto incluye procesamiento de grandes volúmenes de información, análisis predictivo y cómputos científicos. Está orientado a simplificar el despliegue y administración de los paquetes de software.[2]
Las diferentes versiones de los paquetes se administran mediante el sistema de gestión de paquetes *conda*, el cual lo hace bastante sencillo de instalar, correr, y actualizar software de ciencia de datos y aprendizaje automático como puede ser Scikit-team, TensorFlow y SciPy.[3]
La distribución Anaconda es utilizada por 6 millones de usuarios e incluye más de 250 paquetes de ciencia de datos válidos para Windows, Linux y MacOS.

# Anaconda Navigator

https://www.anaconda.com/products/individual

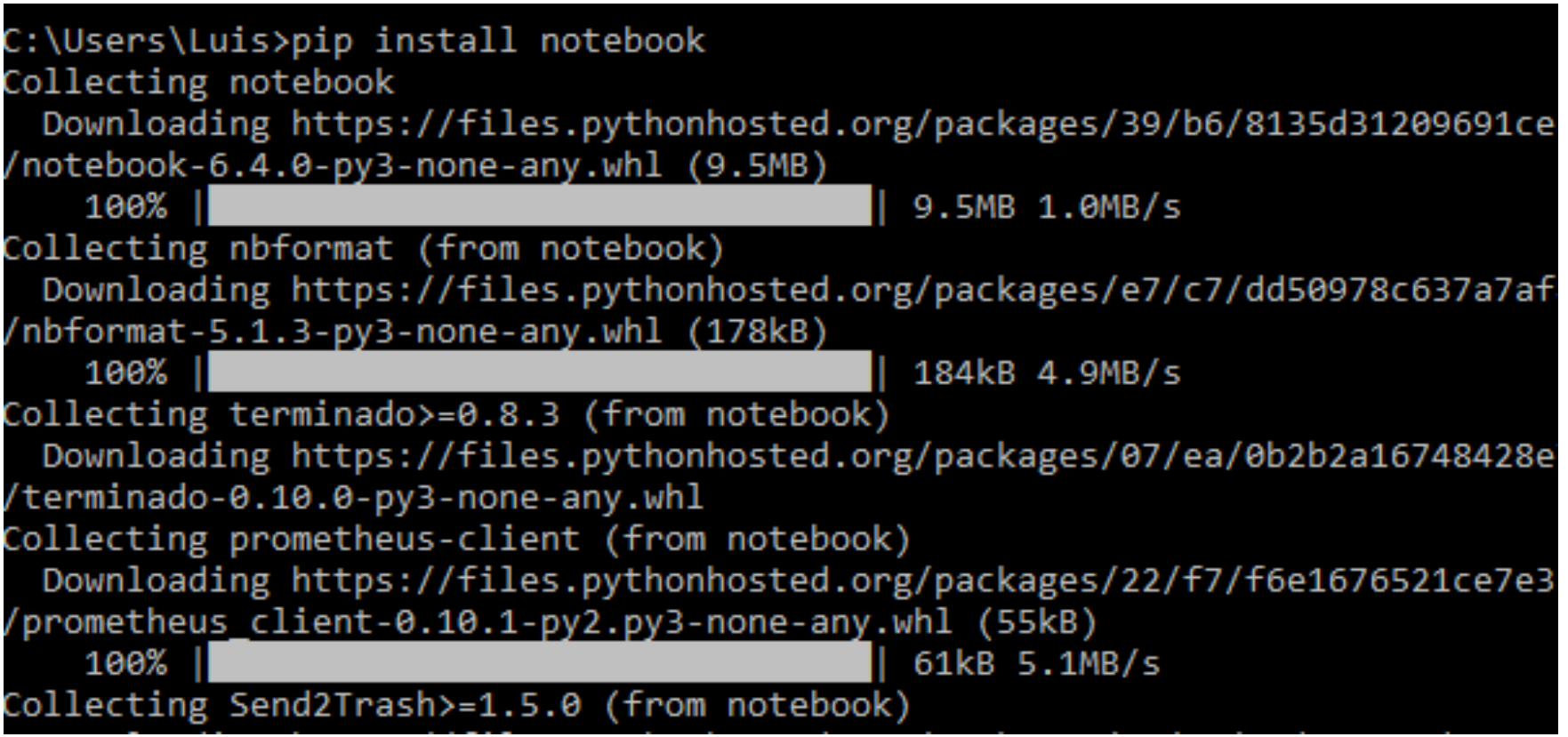


# Pip

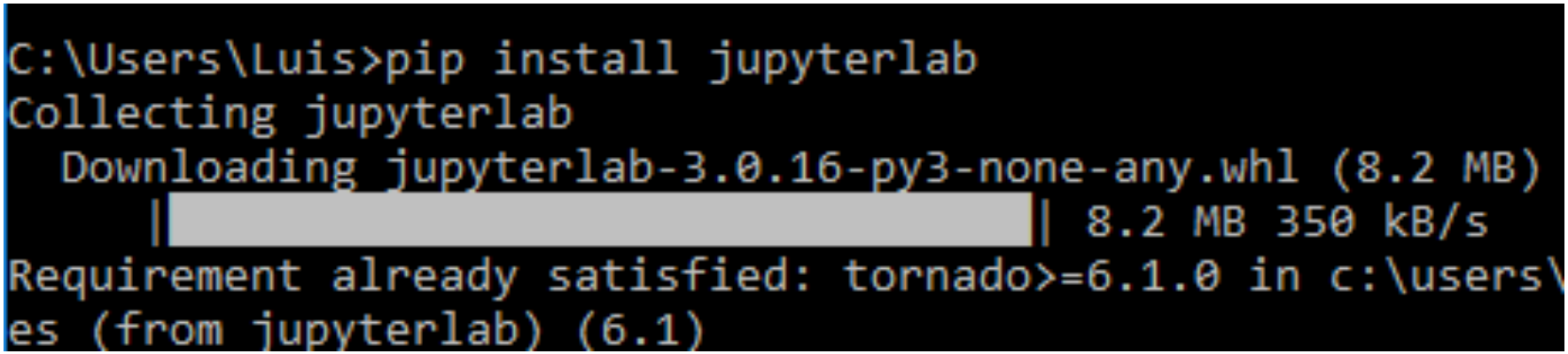
https://es.wikipedia.org/wiki/Pip_(administrador_de_paquetes)

**pip** es un sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en Python. Muchos paquetes pueden ser encontrados en el Python Package Index (PyPI). Python 2.7.9 y posteriores (en la serie Python2), Python 3.4 y posteriores incluyen pip (pip3 para Python3) por defecto.
**pip** es un acrónimo recursivo que se puede interpretar como *Pip Instalador de Paquetes* o *Pip Instalador de Python*.

# Install Jupyter Notebook: pip install notebook



# Option II: Install Jupyter Lab: pip install jupyterlab



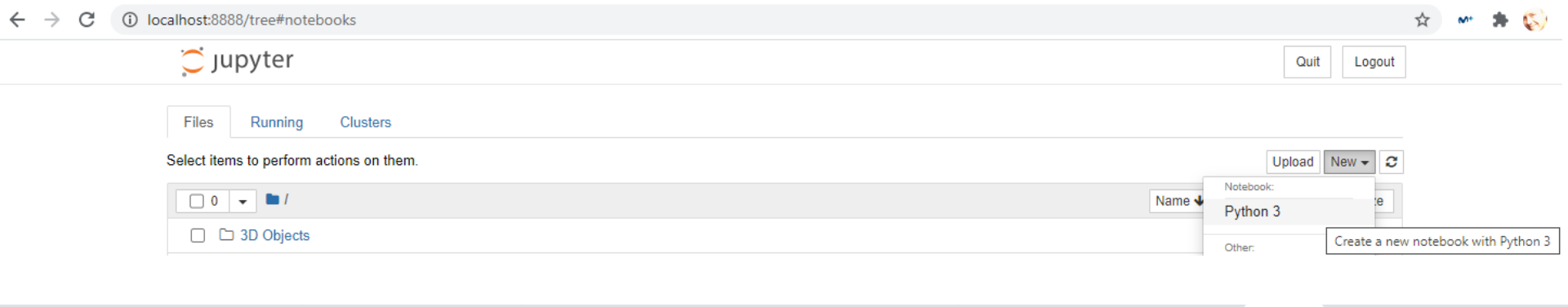
# Install notebook (si falla instalación)

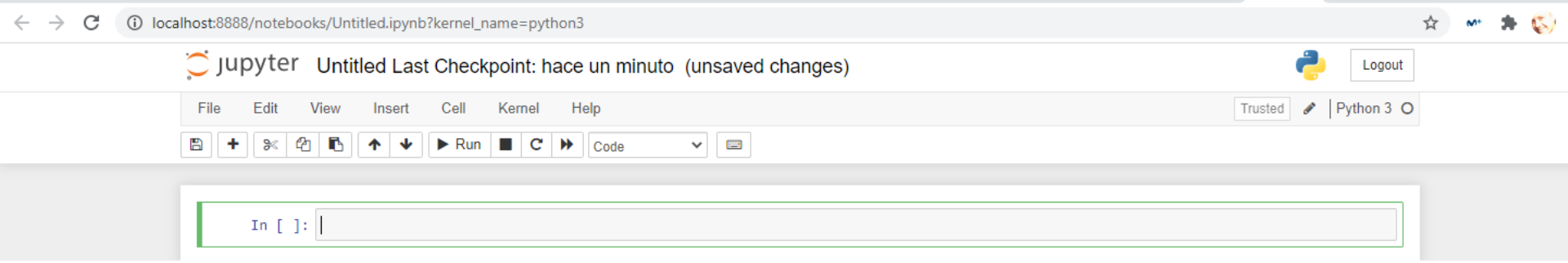
```
C:\Users\Luis>python -m pip install --upgrade pip
Collecting pip
  Downloading https://files.pythonhosted.org/packages/cd/82
/pip-21.1.2-py3-none-any.whl (1.5MB)
    100% |                                | 1.6MB 3.3MB/s
```

```
C:\Users\Luis>pip install notebook
Collecting notebook
  Downloading https://files.pythonhosted.org/packages/39/b6/8135d31209691ce
/notebook-6.4.0-py3-none-any.whl (9.5MB)
    100% |                                | 9.5MB 1.0MB/s
Collecting nbformat (from notebook)
  Downloading https://files.pythonhosted.org/packages/e7/c7/dd50978c637a7af
/nbformat-5.1.3-py3-none-any.whl (178kB)
    100% |                                | 184kB 4.9MB/s
Collecting terminado>=0.8.3 (from notebook)
  Downloading https://files.pythonhosted.org/packages/07/ea/0b2b2a16748428e
/terminado-0.10.0-py3-none-any.whl
Collecting prometheus-client (from notebook)
  Downloading https://files.pythonhosted.org/packages/22/f7/f6e1676521ce7e3
/prometheus_client-0.10.1-py2.py3-none-any.whl (55kB)
    100% |                                | 61kB 5.1MB/s
Collecting Send2Trash>=1.5.0 (from notebook)
```

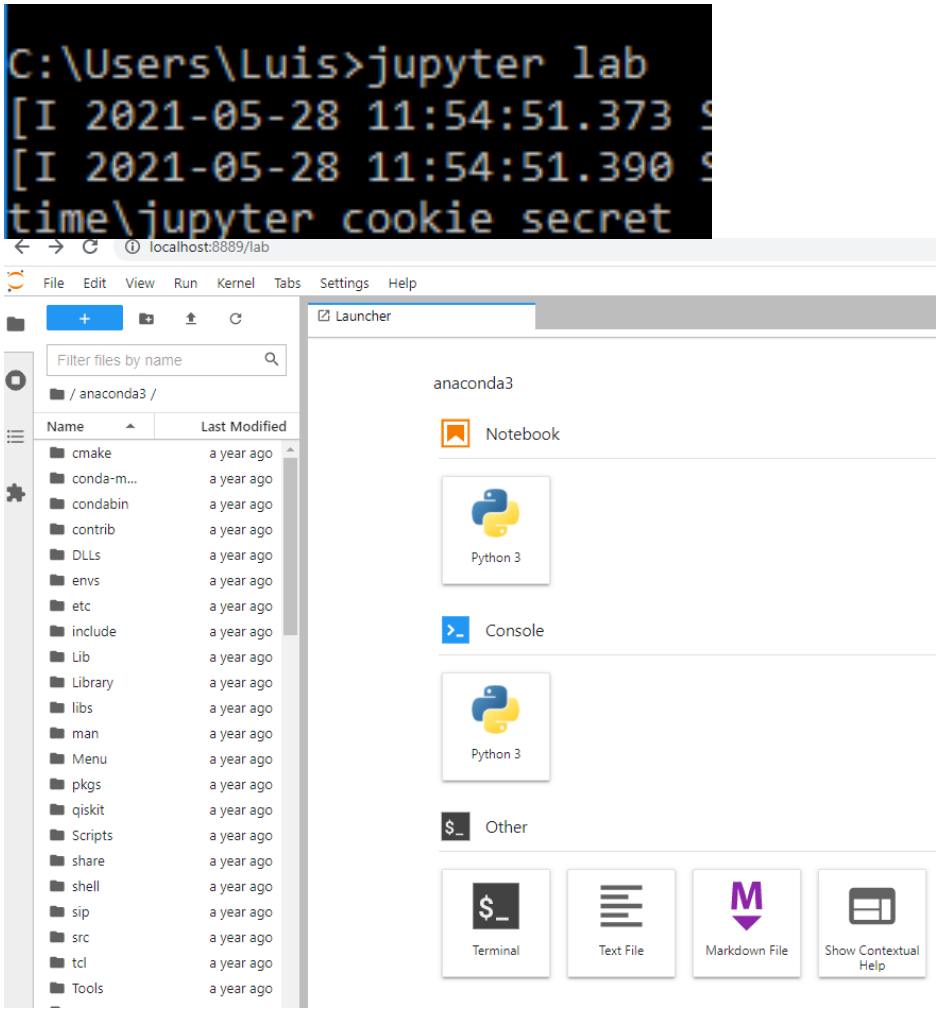
# Run jupyter notebook: jupyter notebook

```
C:\Users\Luis>jupyter notebook
[I 01:30:45.658 NotebookApp] Writing notebook server cookie secret to C:\Users\Luis\AppData\Roaming\jupyter\runtime\note
book_cookie_secret
[I 01:30:46.639 NotebookApp] Serving notebooks from local directory: C:\Users\Luis
[I 01:30:46.640 NotebookApp] Jupyter Notebook 6.4.0 is running at:
[I 01:30:46.640 NotebookApp] http://localhost:8888/?token=6d3886649ae7a55220efd2b92aa15d2874f54c060df4862c
[I 01:30:46.640 NotebookApp]  or http://127.0.0.1:8888/?token=6d3886649ae7a55220efd2b92aa15d2874f54c060df4862c
[I 01:30:46.641 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 01:30:46.730 NotebookApp]
```



# Run jupyter lab:
# jupyter lab



# Run IPython

```
C:\Users\Luis>ipython
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.23.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: ?

IPython -- An enhanced Interactive Python
=========================================

IPython offers a fully compatible replacement for the standard Python
interpreter, with convenient shell features, special commands, command
history mechanism and output results caching.

At your system command line, type 'ipython -h' to see the command line
options available. This document only describes interactive features.

GETTING HELP
------------

Within IPython you have various way to access help:

  ?         -> Introduction and overview of IPython's features (this screen).
  object?   -> Details about 'object'.
  object??  -> More detailed, verbose information about 'object'.
  %quickref -> Quick reference of all IPython specific syntax and magics.
  help      -> Access Python's own help system.
```

# Accessing documentation with ?

```
In [3]: help (len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.


In [4]: len?
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method

In [5]:
```

# Accessing documentation with ?

```
In [5]: L = [1, 2, 3]

In [6]: L.insert?
Signature: L.insert(index, object, /)
Docstring: Insert object before index.
Type:      builtin_function_or_method

In [7]: L?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.
```

# Accessing documentation with ?

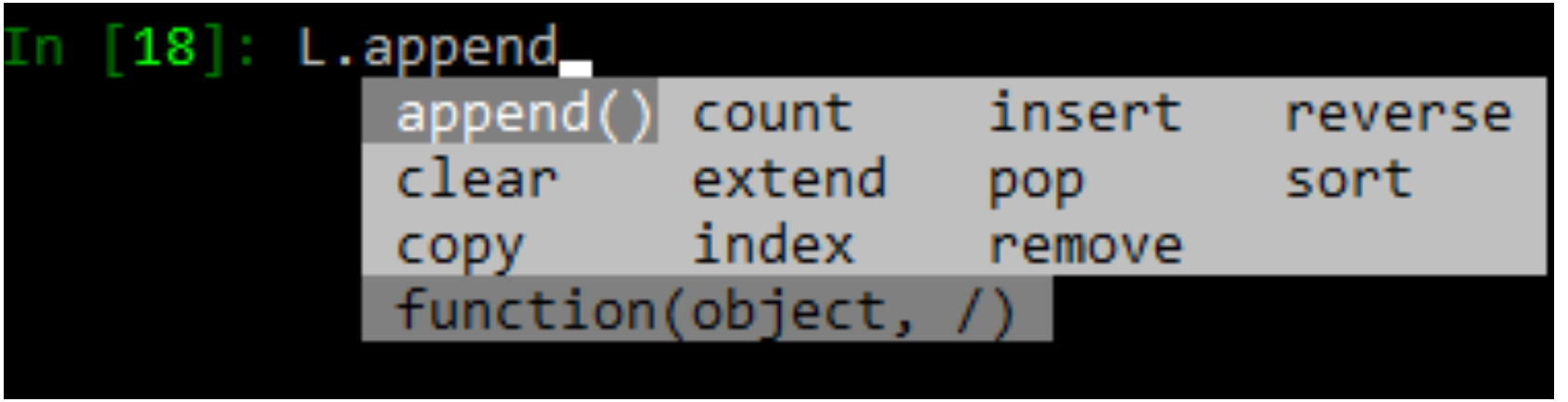
```
In [13]: def square(a):
    ...:     return a ** 2
    ...:

In [14]: square?
Signature: square(a)
Docstring: <no docstring>
File:      c:\users\luis\<ipython-input-13-c61d0fb06e7b>
Type:      function

In [15]: def square(a):
    ...:     """devuelve el cuadrado"""
    ...:     return a ** 2
    ...:
    ...:

In [16]: square?
    ...:
    ...:
Signature: square(a)
Docstring: devuelve el cuadrado
File:      c:\users\luis\<ipython-input-15-d7ced58b74ee>
Type:      function

In [17]:
```

# Accessing source code with ??

```
In [17]: square??
Signature: square(a)
Source:
def square(a):
    """devuelve el cuadrado"""
    return a ** 2
File:      c:\users\luis\<ipython-input-15-d7ced58b74ee>
Type:      function

In [18]:
```

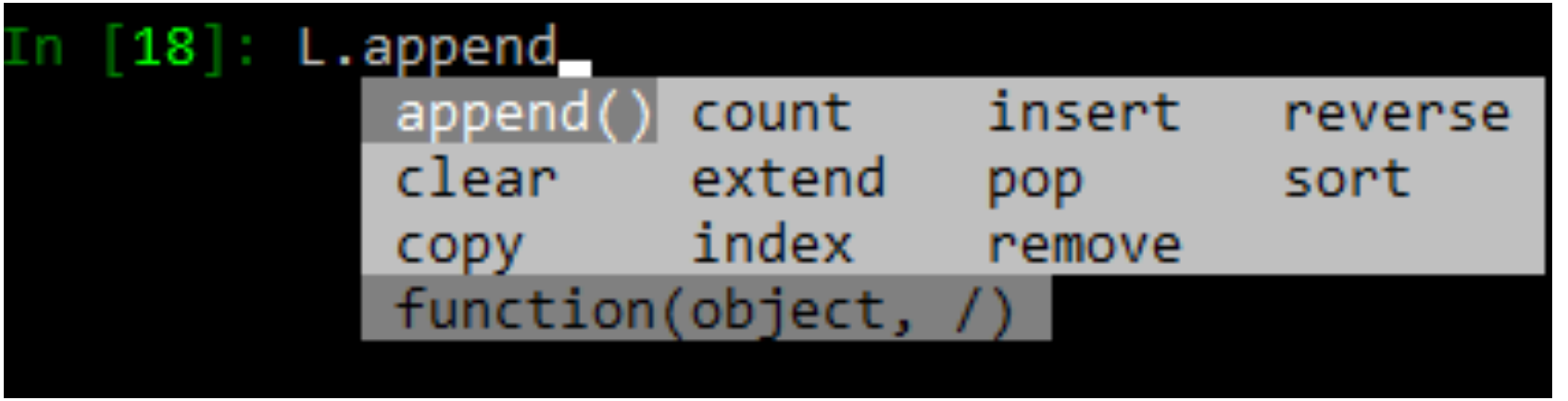
# Exploring Modules with .tab

Write L. + Tab



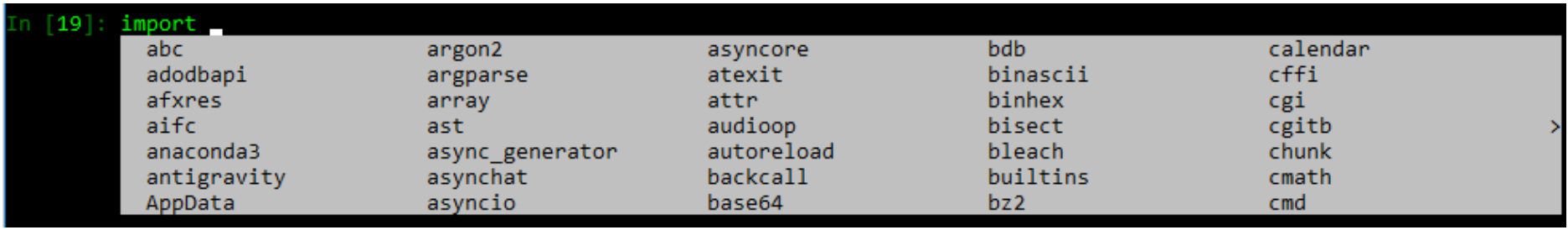
# .tab completion when importing

Write L. + Tab



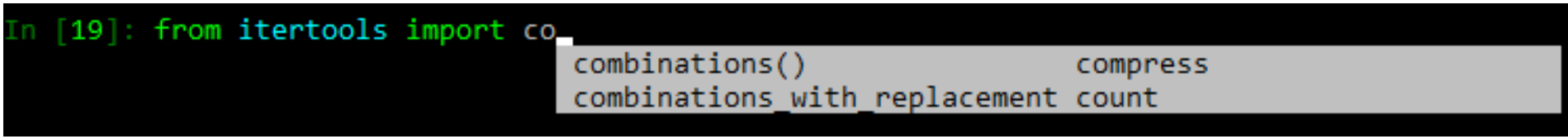
# .tab completion when importing

Import + Tab



# .tab completion when importing

From itertools import co  +TAB



# Wildcard matching

*warning?

```
In [19]: *Warning?
BytesWarning
DeprecationWarning
FutureWarning
ImportWarning
PendingDeprecationWarning
ResourceWarning
RuntimeWarning
SyntaxWarning
UnicodeWarning
UserWarning
Warning
```

# Navigation 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 |

# Test entry 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 |

# Command history 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 |

# Other shortcuts

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

# Ipython Magic Commands
# %run

%run executes a script

```
# file: myscript.py
def square(x):
    return x ** 2
for N in (1, 4):
    print(N, "squared is", square(N))
```

```
In [6]: %run myscript.py
1 squared is 1
2 squared is 4
3 squared is 9
```

# Ipython Magic Commands %timeit

%timeit automatically determine the execution time of the single-line Python statement that follows it

```
%timeit L = [n ** 2 for n in range(1000)]
```

```
In [25]: %timeit L = [n ** 2 for n in range(1000)]
256 µs ± 3.25 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

# Ipython Magic Commands %%timeit

For multi line statements, adding a second `%` sign will turn this into a cell magic that can handle multiple lines of input.

```
In [9]: %%timeit
...: L = []
...: for n in range(1000):
...: L.append(n ** 2)
...:
1000 loops, best of 3: 373 µs per loop
```

# Ipython Magic Commands help

`%magic` To access a general description of available magic functions

```
In [26]: %magic

IPython's 'magic' functions
===========================

The magic function system provides a series of functions which allow you to
control the behavior of IPython itself, plus a lot of system-type
features. There are two kinds of magics, line-oriented and cell-oriented.

Line magics are prefixed with the % character and work much like OS
command-line calls: they get as an argument the rest of the line, where
arguments are passed without parentheses or quotes.  For example, this will
time the given statement::

        %timeit range(1000)

Cell magics are prefixed with a double %%, and they are functions that get as
an argument not only the rest of the line, but also the lines below it in a
separate argument.  These magics are called with two arguments: the rest of the
call line and the body of the cell, consisting of the lines below the first.
For example::

        %%timeit x = numpy.random.randn((100, 100))
        numpy.linalg.svd(x)

will time the execution of the numpy svd routine, running the assignment of x
as part of the setup phase, which is not timed.
```

# Ipython Magic Commands help

`%lsmagic`

For a quick and simple list of all available magic functions

```
In [27]: %lsmagic
Out[27]:
Available line magics:
%alias  %alias_magic  %autoawait  %autocall  %autoindent  %automagic  %bookmark  %cd  %cls  %colors  %conda  %config  %copy
  %cpaste  %ddir  %debug  %dhist  %dirs  %doctest_mode  %echo  %ed  %edit  %env  %gui  %hist  %history  %killbgscripts  %ld
ir  %load  %load_ext  %loadpy  %logoff  %logon  %logstart  %logstate  %logstop  %ls  %lsmagic  %macro  %magic  %matplotlib
 %mkdir  %notebook  %page  %paste  %pastebin  %pdb  %pdef  %pdoc  %pfile  %pinfo  %pinfo2  %pip  %popd  %pprint  %precision
  %prun  %psearch  %psource  %pushd  %pwd  %pycat  %pylab  %quickref  %recall  %rehashx  %reload_ext  %ren  %rep  %rerun  %
reset  %reset_selective  %rmdir  %run  %save  %sc  %set_env  %store  %sx  %system  %tb  %time  %timeit  %unalias  %unload_e
xt  %who  %who_ls  %whos  %xdel  %xmode

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture  %%cmd  %%debug  %%file  %%html  %%javascript  %%js  %%latex  %%markdown  %%perl  %%p
run  %%pypy  %%python  %%python2  %%python3  %%ruby  %%script  %%sh  %%svg  %%sx  %%system  %%time  %%timeit  %%writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

# Ipython Magic Commands help

`%timeit?` Documentation of a magic function

```
In [28]:  %timeit?
Docstring:
Time execution of a Python statement or expression

Usage, in line mode:
  %timeit [-n<N> -r<R> [-t|-c] -q -p<P> -o] statement
or in cell mode:
  %%timeit [-n<N> -r<R> [-t|-c] -q -p<P> -o] setup_code
  code
  code...

Time execution of a Python statement or expression using the timeit
module.  This function can be used both as a line and cell magic:

- In line mode you can time a single-line statement (though multiple
  ones can be chained with using semicolons).

- In cell mode, the statement in the first line is used as setup code
  (executed but not timed) and the body of the cell is timed.  The cell
  body has access to any variables created in the setup code.
```

# In and out objects

```
In [1]: import math
In [2]: math.sin(2)
Out[2]: 0.9092974268256817
In [3]: math.cos(2)
Out[3]: -0.4161468365471424
In [4]: print(In)
['', 'import math', 'math.sin(2)', 'math.cos(2)',
'print(In)']

In [5]: Out  Out[5]:
{2: 0.9092974268256817, 3: -0.4161468365471424}
In [6]: print(In[1])
import math

In [7]: print(Out[2])
0.9092974268256817
```

# Underscore

The standard Python shell contains just one simple shortcut for accessing **previous output**; the variable _ (i.e., a single underscore) is kept updated with the previous output;

```
In [7]: print(_)
0.9092974268256817
```

# Suppressing output: ;

Sometimes you might wish to suppress the output of a statement

(this is perhaps most common with the plotting commands)

```
In [14]: math.sin(2) + math.cos(2);

In [15]: 14 in Out
Out[15]: False
```

# %history

For accessing a batch of previous inputs at once

```
In [16]: %history -n 1-4
    1: import math
    2: math.sin(2)
    3: math.cos(2)
    4: print(In)
```

# IPython shell commands

IPython gives you a syntax for executing shell commands directly from within the IPython terminal.

```
In [30]: echo "hola mundo"
"hola mundo"

In [31]: pwd
Out[31]: 'C:\\Users\\Luis'

In [32]: ls
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 3CF8-3773

 Directorio de C:\Users\Luis

24/02/2021  00:11    <DIR>          .afirma
18/03/2020  12:45    <DIR>          .cache
02/01/2021  17:03    <DIR>          .config
02/01/2021  15:48             1.038 .csslintrc
19/01/2021  04:12    <DIR>          .docker
```

# NumPy
# Numerical Python


Luís Garmendia

# Introduction to NumPy

Numpy gives techniques for effectively loading, storing, and manipulating in-memory data in Python.
The topic is very broad: datasets can come from a wide range of sources and a wide range of formats, including be collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else.
Despite this apparent heterogeneity, it will help us to think of all data fundamentally as **arrays of numbers**.

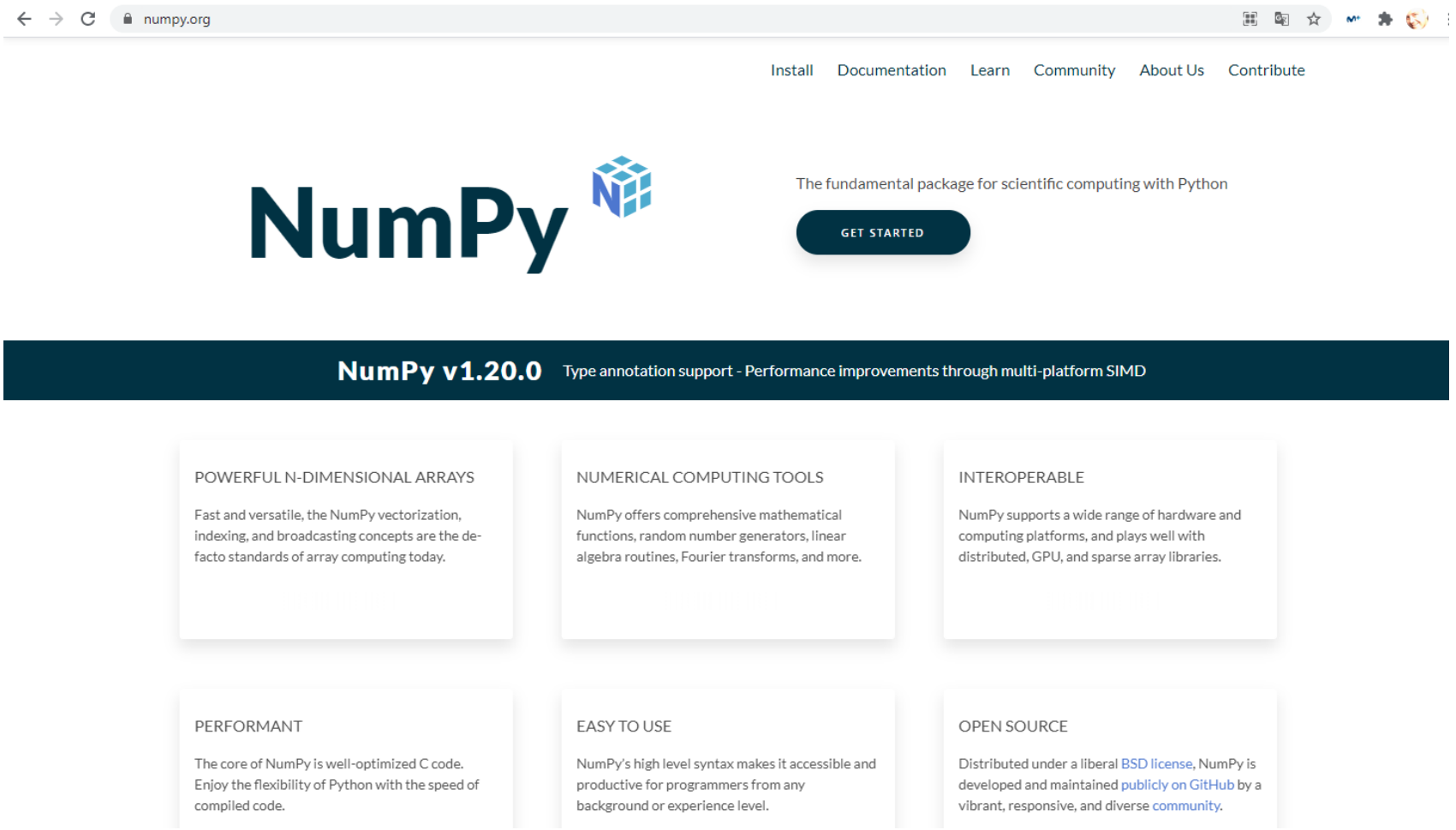
https://numpy.org/

# Introduction to NumPy

For example, **images**–particularly digital images–can be thought of as simply two-dimensional arrays of numbers representing pixel brightness across the area.
**Sound** clips can be thought of as one-dimensional arrays of intensity versus time.
**Text** can be converted in various ways into numerical representations, perhaps binary digits representing the frequency of certain words or pairs of words. No matter what the data are, the first step in making it analyzable will be to transform them into arrays of numbers.

https://numpy.org/

# NumPy (Numerical Python)




https://numpy.org/

# NumPy Ecosystem

## ECOSYSTEM

SCIENTIFIC DOMAINS | ARRAY LIBRARIES | DATA SCIENCE | MACHINE LEARNING | VISUALIZATION

Nearly every scientist working in Python draws on the power of NumPy.

NumPy brings the computational power of languages like C and Fortran to Python, a language much easier to learn and use. With this power comes simplicity: a solution in NumPy is often clear and elegant.

| Quantum Computing | Statistical Computing | Signal Processing | Image Processing | Graphs and Networks | Astronomy Processes | Cognitive Psychology |
|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |
| QuTiP<br>PyQuil<br>Qiskit | Pandas<br>statsmodels<br>Xarray<br>Seaborn | SciPy<br>PyWavelets<br>python-control | Scikit-image<br>OpenCV<br>Mahotas | NetworkX<br>graph-tool<br>igraph<br>PyGSP | AstroPy<br>SunPy<br>SpacePy | PsychoPy |

| Bioinformatics | Bayesian Inference | Mathematical Analysis | Chemistry | Geoscience | Geographic Processing | Architecture & Engineering |
|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |
| BioPython<br>Scikit-Bio<br>PyEnsembl<br>ETE | PyStan<br>PyMC3<br>ArviZ<br>emcee | SciPy<br>SymPy<br>cvxpy<br>FEniCS | Cantera<br>MDAnalysis<br>RDKit | Pangeo<br>Simpeg<br>ObsPy<br>Fatiando a Terra | Shapely<br>GeoPandas<br>Folium | COMPAS<br>City Energy Analyst<br>Sverchok |

https://numpy.org/

# NumPy Ecosystem

## ECOSYSTEM

SCIENTIFIC DOMAINS | ARRAY LIBRARIES | DATA SCIENCE | MACHINE LEARNING | VISUALIZATION

NumPy's API is the starting point when libraries are written to exploit innovative hardware, create specialized array types, or add capabilities beyond what NumPy provides.

| Array Library | | Capabilities & Application areas |
|---|---|---|
| DASK | Dask | Distributed arrays and advanced parallelism for analytics, enabling performance at scale. |
| CuPy | CuPy | NumPy-compatible array library for GPU-accelerated computing with Python. |
| JAX | JAX | Composable transformations of NumPy programs: differentiate, vectorize, just-in-time compilation to GPU/TPU. |
| xarray | Xarray | Labeled, indexed multi-dimensional arrays for advanced analytics and visualization |
| | Sparse | NumPy-compatible sparse array library that integrates with Dask and SciPy's sparse linear algebra. |
| PyTorch | PyTorch | Deep learning framework that accelerates the path from research prototyping to production deployment. |
| TensorFlow | TensorFlow | An end-to-end platform for machine learning to easily build and deploy ML powered applications. |
| mxnet | MXNet | Deep learning framework suited for flexible research prototyping and production. |
| ARROW | Arrow | A cross-language development platform for columnar in-memory data and analytics. |
| xtensor | xtensor | Multi-dimensional arrays with broadcasting and lazy computing for numerical analysis. |

https://numpy.org/

# NumPy Ecosystem

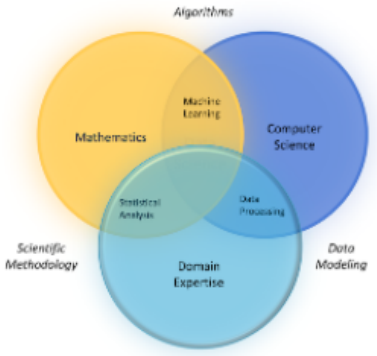
ECOSYSTEM

SCIENTIFIC DOMAINS ARRAY LIBRARIES DATA SCIENCE MACHINE LEARNING VISUALIZATION


NumPy lies at the core of a rich ecosystem of data science libraries. A typical exploratory data science workflow might look like:

**Extract, Transform, Load:** Pandas, Intake, PyJanitor
**Exploratory analysis:** Jupyter, Seaborn, Matplotlib, Altair
**Model and evaluate:** scikit-learn, statsmodels, PyMC3, spaCy
**Report in a dashboard:** Dash, Panel, Voila

For high data volumes, Dask and Ray are designed to scale. Stable deployments rely on data versioning (DVC), experiment tracking (MLFlow), and workflow automation (Airflow and Prefect).




https://numpy.org/

# NumPy Ecosystem

## ECOSYSTEM

SCIENTIFIC DOMAINS ARRAY LIBRARIES DATA SCIENCE MACHINE LEARNING VISUALIZATION

*Source: Google AI Blog*

NumPy forms the basis of powerful machine learning libraries like scikit-learn and SciPy. As machine learning grows, so does the list of libraries built on NumPy. TensorFlow's deep learning capabilities have broad applications — among them speech and image recognition, text-based applications, time-series analysis, and video detection. PyTorch, another deep learning library, is popular among researchers in computer vision and natural language processing. MXNet is another AI package, providing blueprints and templates for deep learning.

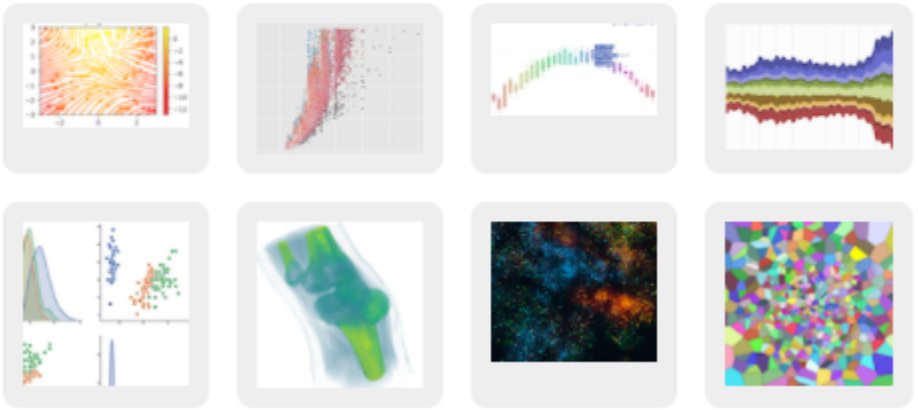
Statistical techniques called ensemble methods such as binning, bagging, stacking, and boosting are among the ML algorithms implemented by tools such as XGBoost, LightGBM, and CatBoost — one of the fastest inference engines. Yellowbrick and Eli5 offer machine learning visualizations.

https://numpy.org/

# NumPy Ecosystem

## ECOSYSTEM

SCIENTIFIC DOMAINS ARRAY LIBRARIES DATA SCIENCE MACHINE LEARNING VISUALIZATION



NumPy is an essential component in the burgeoning Python visualization landscape, which includes Matplotlib, Seaborn, Plotly, Altair, Bokeh, Holoviz, Vispy, Napari, and PyVista, to name a few.

NumPy's accelerated processing of large arrays allows researchers to visualize datasets far larger than native Python could handle.

https://numpy.org/

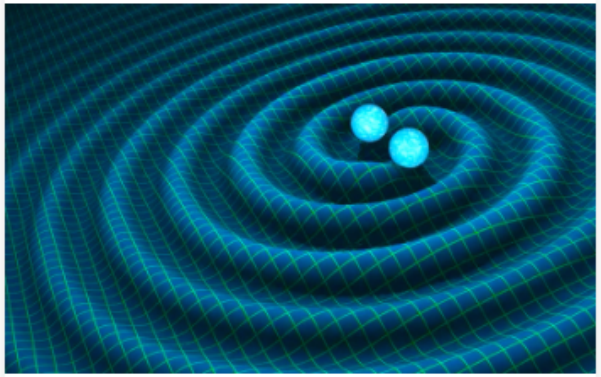
# NumPy use cases

## CASE STUDIES

### FIRST IMAGE OF A BLACK HOLE

How NumPy, together with libraries like SciPy and Matplotlib that depend on NumPy, enabled the Event Horizon Telescope to produce the first ever image of a black hole

### DETECTION OF GRAVITATIONAL WAVES



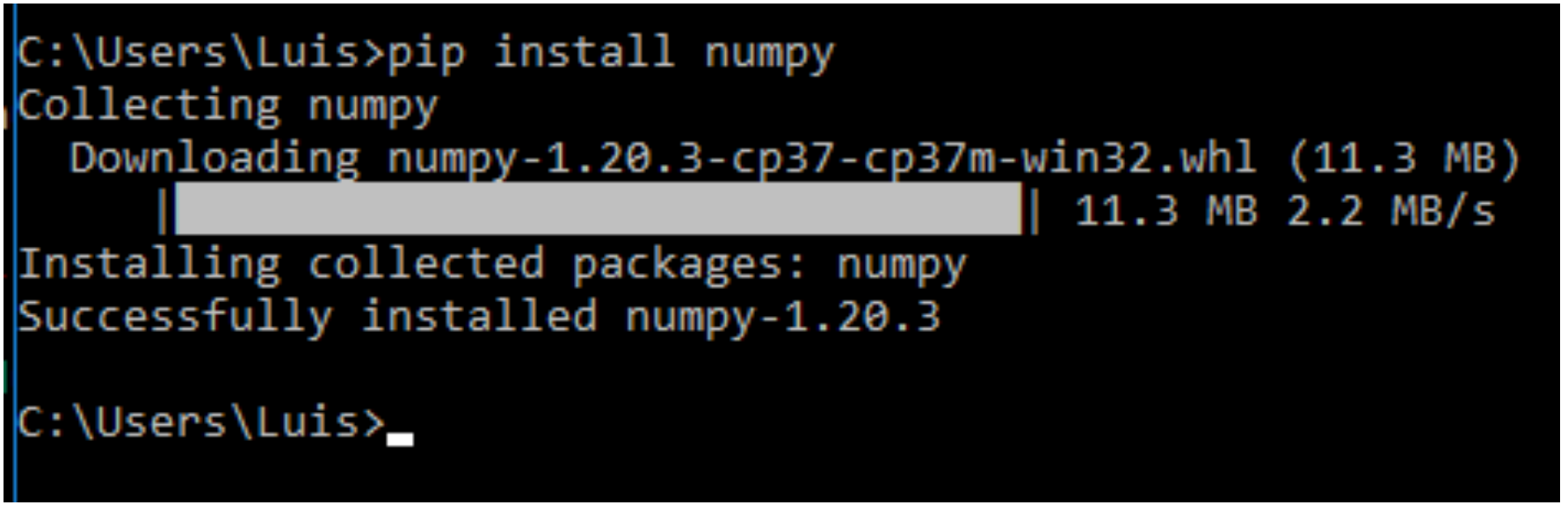
In 1916, Albert Einstein predicted gravitational waves; 100 years later their existence was confirmed by LIGO scientists using NumPy.

### SPORTS ANALYTICS

Cricket Analytics is changing the game by improving player and team performance through statistical modelling and predictive analytics. NumPy enables many of these analyses.

https://numpy.org/

# NumPy Installation
# pip install numpy




https://numpy.org/

# NumPy
# numpy.__version__

```
[38]: import numpy
 ...: numpy.__version__
[38]: '1.20.3'

[39]: import numpy as np

[40]:
```

https://numpy.org/

# NumPy
# np +TAB

```
In [40]: np._
  abs                   add_newdoc()          allclose()            amin()
  absolute              add_newdoc_ufunc()    ALLOW_THREADS         angle()
  add                   alen()                alltrue()             any()
  add_docstring()       all()                 amax()                append()
```

https://numpy.org/

# Array manipulations

- *Attributes of arrays*: Determining the size, shape, memory consumption, and data types of arrays
- *Indexing of arrays*: Getting and setting the value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

https://numpy.org/

# Numpy random numbers

- We'll use NumPy's random number generator, which we will *seed* with a set value.

```
import numpy as np
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
x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional
```

https://numpy.org/

# NumPy array attributes

Each array has attributes **ndim** (the number of dimensions), **shape** (the size of each dimension), and **size** (the total size of the array):

```
In [43]: print("x3 ndim: ", x3.ndim)
    ...: print("x3 shape:", x3.shape)
    ...: print("x3 size: ", x3.size)
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60

In [44]: for i in x3:
    ...:     print(i)
    ...:
[[5 0 3 3 7]
 [9 3 5 2 4]
 [7 6 8 8 1]
 [6 7 7 8 1]]
[[5 9 8 9 4]
 [3 0 3 5 0]
 [2 3 8 1 3]
 [3 3 7 0 1]]
[[9 9 0 4 7]
 [3 2 7 2 0]
 [0 4 5 5 6]
 [8 4 1 4 9]]
```

```
In [45]: x3
Out[45]:
array([[[5, 0, 3, 3, 7],
        [9, 3, 5, 2, 4],
        [7, 6, 8, 8, 1],
        [6, 7, 7, 8, 1]],

       [[5, 9, 8, 9, 4],
        [3, 0, 3, 5, 0],
        [2, 3, 8, 1, 3],
        [3, 3, 7, 0, 1]],

       [[9, 9, 0, 4, 7],
        [3, 2, 7, 2, 0],
        [0, 4, 5, 5, 6],
        [8, 4, 1, 4, 9]]])
```

# NumPy array attributes

the **dtype** attribute is the data type of the array.
Other attributes include **itemsize**, which lists the size (in bytes) of each array element, and **nbytes**, which lists the total size (in bytes) of the array.

```
In [46]: print("dtype:", x3.dtype)
dtype: int32

In [47]: print("itemsize:", x3.itemsize, "bytes")
    ...: print("nbytes:", x3.nbytes, "bytes")
itemsize: 4 bytes
nbytes: 240 bytes
```

# Array Indexing
# Accessing single elements

Just as with Python lists:

```
In [49]: x1 = np.random.randint(10, size=6)

In [50]: x1
Out[50]: array([8, 1, 1, 7, 9, 9])

In [51]: x1[0]
Out[51]: 8

In [52]: x1[4]
Out[52]: 9

In [53]: x1[-1]
Out[53]: 9

In [54]: x1[-3]
Out[54]: 7

In [55]: x3[2][2][3]
Out[55]: 5
```

# Array Slicing
# Accessing subarrays

We can also access subarrays with the *slice* notation, marked by the colon (:)

```
In [56]: x = np.arange(10)

In [57]: x
Out[57]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [58]: x[:5]
Out[58]: array([0, 1, 2, 3, 4])

In [59]: x[5:]
Out[59]: array([5, 6, 7, 8, 9])

In [60]: x[4:7]
Out[60]: array([4, 5, 6])
```

# Array Slicing
# step

x[start:stop:step]

```
In [61]: x[::2]
Out[61]: array([0, 2, 4, 6, 8])

In [62]: x[::3]
Out[62]: array([0, 3, 6, 9])

In [63]: x[1::2]
Out[63]: array([1, 3, 5, 7, 9])
```

# Array Slicing bidimensional

x[start:stop:step]

```
In [64]: x2 = np.random.randint(10, size=(3, 4))

In [65]: x2
Out[65]:
array([[3, 6, 7, 2],
       [0, 3, 5, 9],
       [4, 4, 6, 4]])

In [66]: x2[:2,:3]
Out[66]:
array([[3, 6, 7],
       [0, 3, 5]])

In [67]: x2[:3,::2]
Out[67]:
array([[3, 7],
       [0, 5],
       [4, 6]])
```

# Array Slicing
# views (no copies)

Array slices 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.

```
In [71]: x2
Out[71]:
array([[3, 6, 7, 2],
       [0, 3, 5, 9],
       [4, 4, 6, 4]])

In [72]: x2v = x2[:2,:2]

In [73]: x2v
Out[73]:
array([[3, 6],
       [0, 3]])

In [74]: x2v[0,0] = 99

In [75]: x2
Out[75]:
array([[99,  6,  7,  2],
       [ 0,  3,  5,  9],
       [ 4,  4,  6,  4]])
```

# Array Slicing
# copies of arrays: .copy()

It is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the copy() method.

```
In [76]: x2c = x2[:2,:2].copy()

In [77]: x2c
Out[77]:
array([[99,  6],
       [ 0,  3]])

In [78]: x2c[0][0]= 44

In [79]: x2c
Out[79]:
array([[44,  6],
       [ 0,  3]])

In [80]: x2
Out[80]:
array([[99,  6,  7,  2],
       [ 0,  3,  5,  9],
       [ 4,  4,  6,  4]])
```

# Array reshape

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the reshape method

```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

```
In [81]: grid = np.arange(1, 10).reshape((3, 3))
    ...: print(grid)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

# Array concatenation

Concatenation is primarily accomplished using the routine **np.concatenate**

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

```
In [82]: x = np.array([1, 2, 3])
    ...: y = np.array([3, 2, 1])
    ...: np.concatenate([x, y])
Out[82]: array([1, 2, 3, 3, 2, 1])
```

# Array concatenation

It can also be used for two-dimensional arrays:

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

```
In [83]: grid = np.array([[1, 2, 3], [4, 5, 6]])
    ...: np.concatenate([grid, grid])
Out[83]:
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

# Array splitt

The opposite of concatenation is splitting, which is implemented by the functions np.split

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

```
In [84]: x = [1, 2, 3, 99, 99, 3, 2, 1]^M
    ...: x1, x2, x3 = np.split(x, [3, 5])^
    ...: print(x1, x2, x3)
[1 2 3] [99 99] [3 2 1]
```

# Universal Functions: ufuncs

Computation on NumPy arrays can be very fast, or it can be very slow. The key to making it fast is to use ***vectorized* operations**, generally implemented through NumPy's *universal functions* (ufuncs)
This can be accomplished by simply performing an operation on the array, which will then be applied to each element. This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

```
 np.arange(5) / np.arange(1, 6)
x = np.arange(9).reshape((3, 3))
2 ** x
```

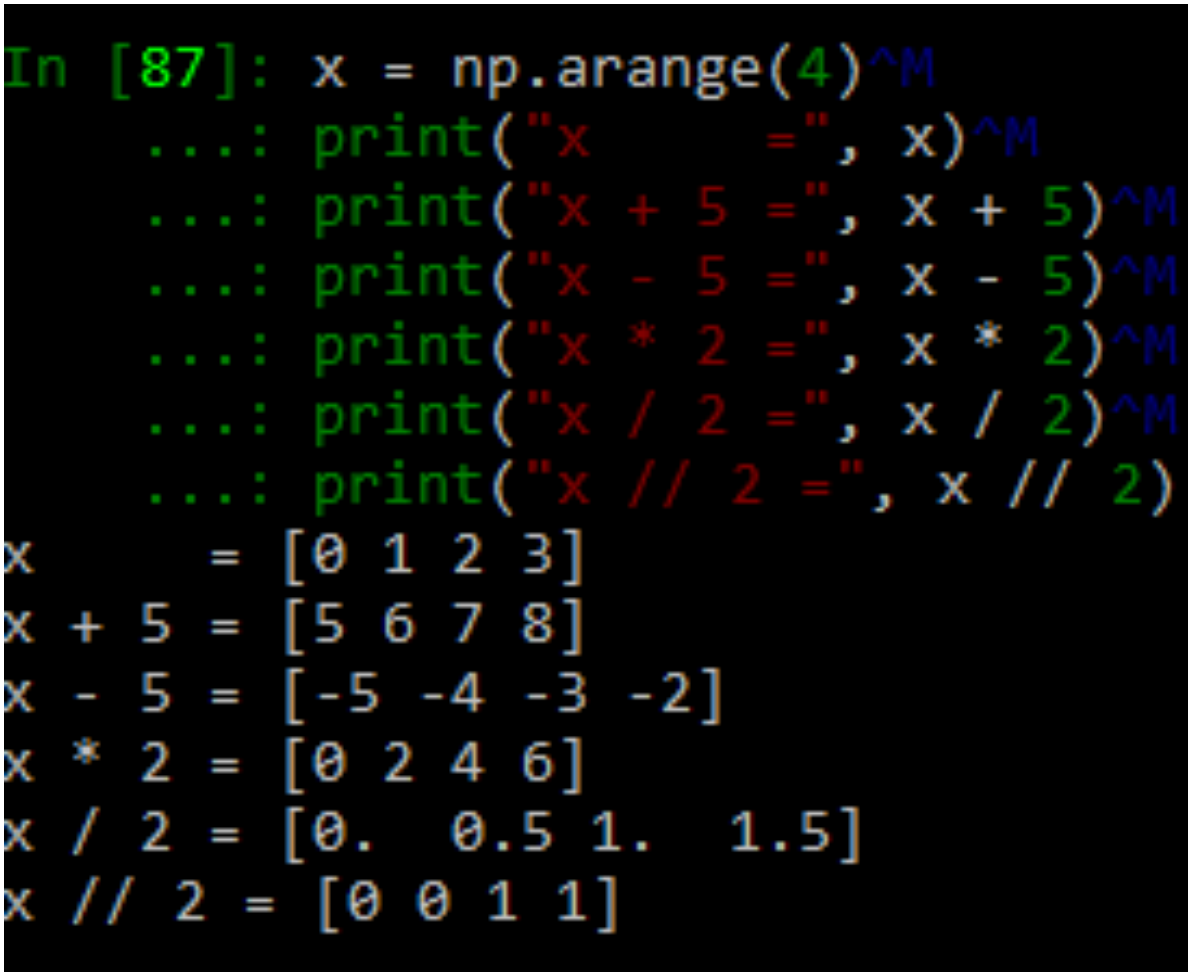
# Universal Functions: ufuncs

```
np.arange(5) / np.arange(1, 6)

x = np.arange(9).reshape((3, 3))
2 ** x
```

```
In [85]: np.arange(5) / np.arange(1, 6)
Out[85]: array([0.        , 0.5       , 0.66666667, 0.75      , 0.8       ])

In [86]: x = np.arange(9).reshape((3, 3))
    ...: 2 ** x
Out[86]:
array([[  1,   2,   4],
       [  8,  16,  32],
       [ 64, 128, 256]], dtype=int32)
```

# Universal Functions: 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
```



# Universal Functions: ufuncs

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

```
In [88]: print("-x     = ", -x)^M
    ...: print("x ** 2 = ", x ** 2)
    ...: print("x % 2  = ", x % 2)
-x     =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2  =  [0 1 0 1]
```

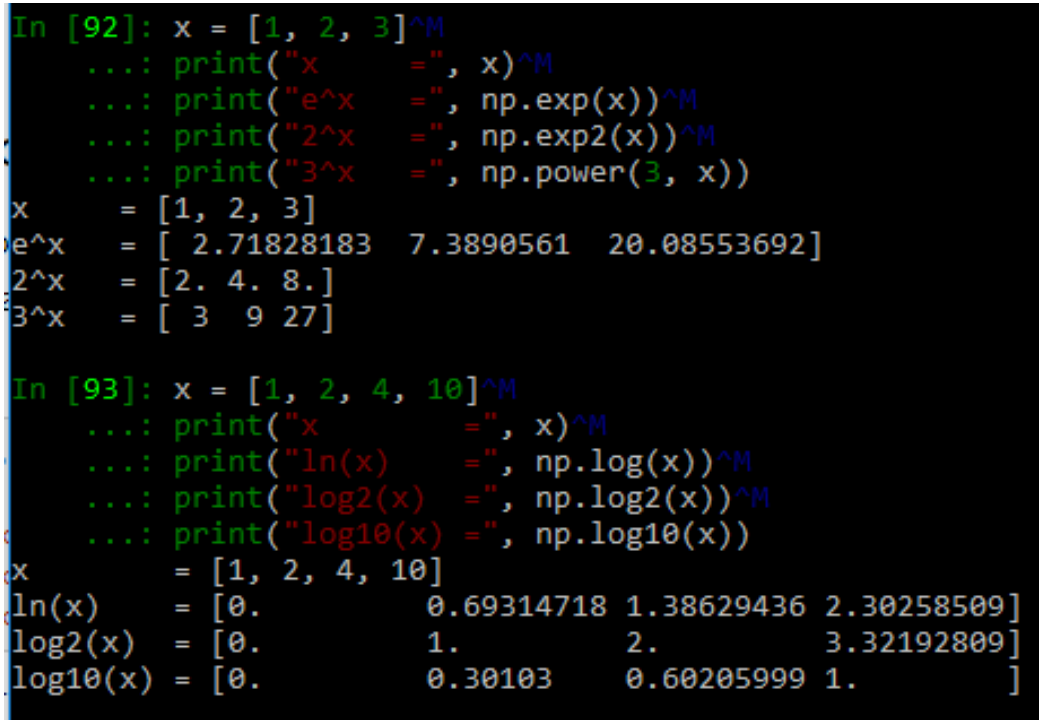
# Universal Functions: ufuncs

```
x = np.array([-2, -1, 0, 1, 2])
np.abs(x)

theta = np.linspace(0, np.pi, 3)
print("theta = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
```

```
In [90]: x = np.array([-2, -1, 0, 1, 2]) ^M
    ...: np.abs(x) ^M
    ...:
Out[90]: array([2, 1, 0, 1, 2])

In [91]: print("theta      = ", theta)^M
    ...: print("sin(theta) = ", np.sin(theta))^M
    ...: print("cos(theta) = ", np.cos(theta))^M
    ...: print("tan(theta) = ", np.tan(theta))
theta      =  [0.         1.57079633 3.14159265]
sin(theta) =  [0.0000000e+00 1.0000000e+00 1.2246468e-16]
cos(theta) =  [ 1.000000e+00  6.123234e-17 -1.000000e+00]
tan(theta) =  [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

# Universal Functions: ufuncs

```
x = [1, 2, 3]
print("x =", x)
print("e^x =", np.exp(x))
print("2^x =", np.exp2(x))
print("3^x =", np.power(3, x))

x = [1, 2, 4, 10]
print("x =", x)
print("ln(x) =", np.log(x))
print("log2(x) =", np.log2(x))
print("log10(x) =", np.log10(x))
```



# Specialized ufuncs

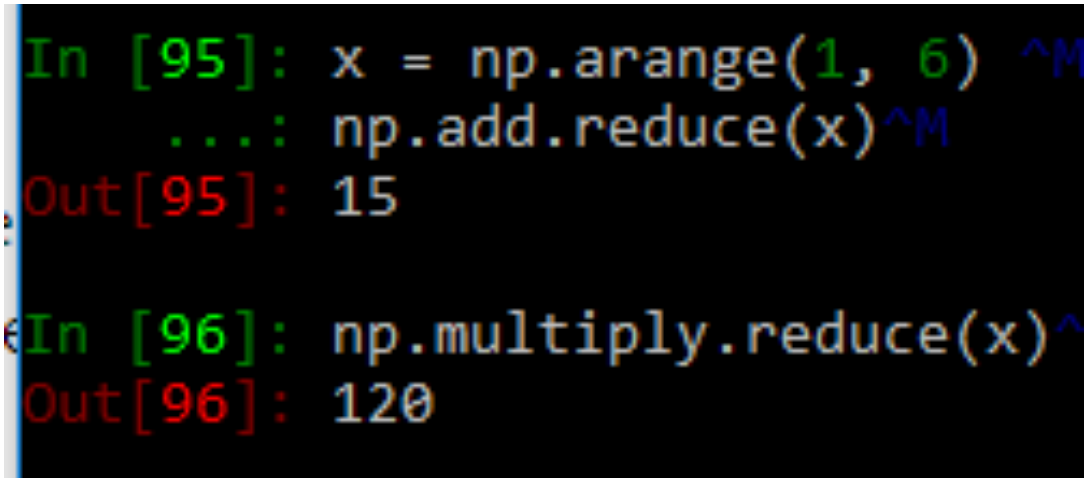
NumPy has many more ufuncs available, including hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. A look through the NumPy documentation reveals a lot of interesting functionality. Another excellent source for more specialized and obscure ufuncs is the submodule scipy.special

```
from scipy import special
x = [1, 5, 10]
print("gamma(x) =", special.gamma(x))
print("ln|gamma(x)| =", special.gammaln(x))
print("beta(x, 2) =", special.beta(x, 2))
```

# Aggregates .reduce

For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we'd like to ***reduce*** an array with a particular operation, we can use the reduce method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

```
x = np.arange(1, 6)
np.add.reduce(x)
np.multiply.reduce(x)
```



# Aggregates .acumulate

If we'd like to store all the intermediate results of the computation, we can instead use accumulate

np.add.accumulate(x)

np.multiply.accumulate(x)

```
In [97]: np.add.accumulate(x)
Out[97]: array([ 1,  3,  6, 10, 15], dtype=int32)

In [98]: np.multiply.accumulate(x)
Out[98]: array([  1,   2,   6,  24, 120], dtype=int32)
```

# Outer products .outer

Any ufunc can compute the output of all pairs of two different inputs using the outer method.
This allows you, in one line, to do things like create a multiplication table

```
x = np.arange(1, 6)
np.multiply.outer(x, x)
```

```
In [99]: x = np.arange(1, 6)^M
    ...: np.multiply.outer(x, x)
Out[99]:
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]])
```

# Aggregations sum

```
import numpy as np

L = np.random.random(100)
np.sum(L)


big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

```
In [100]: big_array = np.random.rand(1000000)^M
     ...: %timeit sum(big_array)^M
     ...: %timeit np.sum(big_array)
121 ms ± 1.88 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
1.05 ms ± 39.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

# Aggregations

For min, max, sum, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself

```
print(big_array.min(), big_array.max(), big_array.sum())
```

```
In [101]: print(big_array.min(), big_array.max(), big_array.sum())
1.4057692298008462e-06 0.9999994392723005 500209.12067471276
```

# Multi dimensional aggregations axis

One common type of aggregation operation is an aggregate along a row or column.

```
 M = np.random.random((3, 4))
print(M)
M.sum()
#min of every column
M.min(axis=0)
#min of every row
M.max(axis=1)
```

```
In [103]:  M = np.random.random((3, 4)) ^M
     ...: print(M)
[[7.84029145e-01 7.50099011e-04 5.08583464e-01 4.44828020e-01]
 [2.17253314e-02 4.65670506e-01 9.04110146e-01 2.54502952e-01]
 [1.58300390e-01 9.84526510e-01 4.36080488e-01 7.44398739e-02]]

In [104]: M.sum()
Out[104]: 5.0375469245872555

In [105]: M.min(axis=0)
Out[105]: array([0.02172533, 0.0007501 , 0.43608049, 0.07443987])

In [106]: M.min(axis=0)
Out[106]: array([0.02172533, 0.0007501 , 0.43608049, 0.07443987])
```

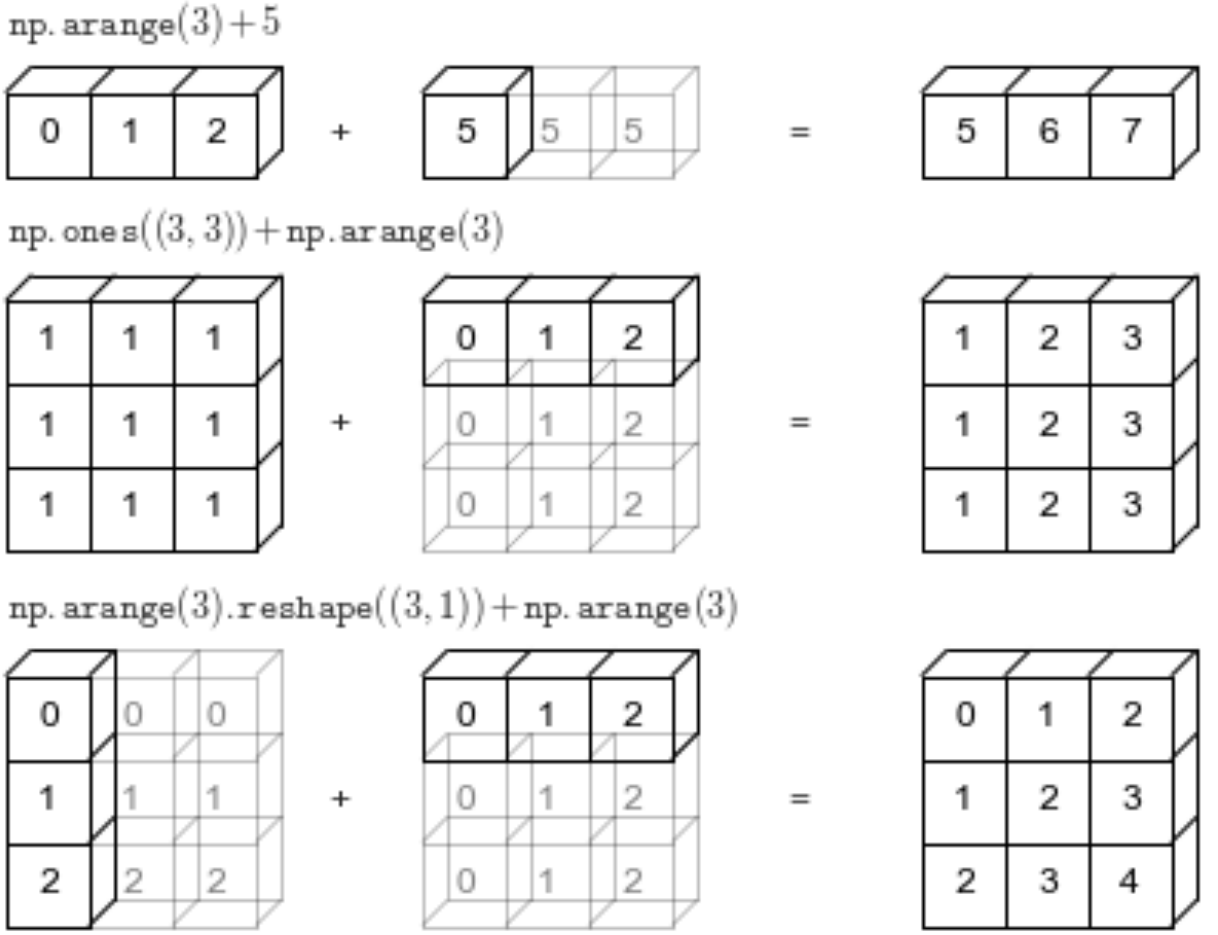
# Broadcasting

Broadcasting is simply a set of rules for applying binary ufuncs (e.g., addition, subtraction, multiplication, etc.) on arrays of different sizes.

```
import numpy as np
a = np.array([0, 1, 2])
a + 5

M = np.ones((3, 3))
M + a
```

```
In [107]: a = np.array([0, 1, 2])

In [108]: a+5
Out[108]: array([5, 6, 7])

In [109]: M = np.ones((3, 3))

In [110]: M
Out[110]:
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])

In [111]: M + a
Out[111]:
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

# Broadcasting



# Broadcasting
# np.newaxis

```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]
print(a)
print(b)

a + b
```

```
In [112]: a = np.arange(3)^M
     ...: b = np.arange(3)[:, np.newaxis]^M
     ...: ^M
     ...: print(a)^M
     ...: print(b)
[0 1 2]
[[0]
 [1]
 [2]]

In [113]: a+b
Out[113]:
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

# Comparison, masks and Boolean logic

Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold.
In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

# Comparison, masks and Boolean logic

NumPy also implements comparison operators such as < (less than) and > (greater than) as element-wise ufuncs. The result of these comparison operators is always an array with a Boolean data type.

```
x = np.array([1, 2, 3, 4, 5])
x < 3
x <= 3
x == 3
x != 3
(2 * x) == (x ** 2)
```

```
In [119]: x = np.array([1, 2, 3, 4, 5])
     ...: x < 3
Out[119]: array([ True,  True, False, False, False])

In [120]: x <= 3
Out[120]: array([ True,  True,  True, False, False])

In [121]: x == 3
Out[121]: array([False, False,  True, False, False])

In [122]: x != 3
Out[122]: array([ True,  True, False,  True,  True])

In [123]: (2 * x) == (x ** 2)
Out[123]: array([False,  True, False, False, False])
```

# Counting entries

To count the number of True entries in a Boolean array, np.count_nonzero is useful:

```
np.count_nonzero(x < 6)
```

We see that there are eight array entries that are less than 6. Another way to get at this information is to use np.sum; in this case, False is interpreted as 0, and True is interpreted as 1:

```
np.sum(x < 6)
```

```
[124]: np.count_nonzero(x < 6)
[124]: 5

[125]: np.sum(x < 6)
[125]: 5
```

# .any and .all

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) np.any or np.all

np.any(x > 8)

np.all(x < 10)

```
[128]: np.any(x > 8)
[128]: False

[129]: np.all(x < 10)
[129]: True
```

# Boolean operators

| Operator | Equivalent ufunc | Operator | Equivalent ufunc |
|---|---|---|---|
| & | np.bitwise_and | \| | np.bitwise_or |
| ^ | np.bitwise_xor | ~ | np.bitwise_not |

```
np.sum((inches > 0.5) & (inches < 1))
```

```
np.sum(~( (inches <= 0.5) | (inches >= 1) ))
```

# Masks

To *select* these values from the array, we can simply index on this Boolean array; this is known as a *masking* operation:

```
[135]: x
[135]: array([1, 2, 3, 4, 5])

[136]: x < 5
[136]: array([ True,  True,  True,  True, False])

[137]: x[x<5]
[137]: array([1, 2, 3, 4])

[138]: x[x<3]
[138]: array([1, 2])
```

# Fancy indexing

Fancy indexing means passing an array of indices to access multiple array elements at once.

```
import numpy as np
rand = np.random.RandomState(42)
x = rand.randint(100, size=10)
print(x)
```

```
[x[3], x[7], x[2]]
```

or

```
ind = [3, 7, 4]
x[ind]
```

```
In [140]: import numpy as np
     ...: rand = np.random.RandomState(42
     ...:
     ...: x = rand.randint(100, size=10)
     ...: print(x)
[51 92 14 71 60 20 82 86 74 74]

In [141]: ind = [3, 7, 4]
     ...: x[ind]
Out[141]: array([71, 86, 60])
```

# Combined indexing

We can combine fancy and simple indices:

```
print(X)
[[ 0, 1, 2, 3] ,
[ 4, 5, 6, 7],
[ 8, 9, 10, 11]]
```

```
X[2, [2, 0, 1]]
```

# Sorting np.sort

To return a sorted version of the array without modifying the input, you can use np.sort:

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

If you prefer to sort the array in-place, you can instead use the sort method of arrays:

```
x.sort()
print(x)
```

```
In [143]: x = np.array([2, 1, 4, 3, 5])

In [144]: x.sort()
     ...: print(x)
[1 2 3 4 5]
```

# argsort

A related function is argsort, which instead returns the *indices* of the sorted elements:

```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)

x[i]
```

```
In [145]: x = np.array([2, 1, 4, 3, 5])
     ...: i = np.argsort(x)
     ...: print(i)
[1 0 3 2 4]

In [146]: x[i]
Out[146]: array([1, 2, 3, 4, 5])
```

# Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument.

```
rand = np.random.RandomState(42)
X = rand.randint(0, 10, (4, 6))
print(X)
```

```
np.sort(X, axis=0)
```

```
np.sort(X, axis=1)
```

```
In [147]: rand = np.random.RandomState(42)
     ...: X = rand.randint(0, 10, (4, 6))
     ...: print(X)
[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]

In [148]: np.sort(X, axis=0)
Out[148]:
array([[2, 1, 4, 0, 1, 5],
       [5, 2, 5, 4, 3, 7],
       [6, 3, 7, 4, 6, 7],
       [7, 6, 7, 4, 9, 9]])

In [149]: np.sort(X, axis=1)
Out[149]:
array([[3, 4, 6, 6, 7, 9],
       [2, 3, 4, 6, 7, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 5, 9]])
```

# Partitioning

Sometimes we're not interested in sorting the entire array, but simply want to find the *k* smallest values in the array. NumPy provides this in the np.partition function

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

```
array([2, 1, 3, 4, 6, 5, 7])
```

Note that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

# Structured arrays

While often our data can be well represented by a homogeneous array of values

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

NumPy can handle this through structured arrays, which are arrays with compound data types.

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

# Structured arrays

Now that we've created an empty container array, we can fill the array with our lists of values:

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

# Get all names data['name']
array(['Alice', 'Bob', 'Cathy', 'Doug'], dtype='<U10')
```

# Structured arrays

The handy thing with structured arrays is that you can now refer to values either by index or by name:

```
# Get all names data['name']
array(['Alice', 'Bob', 'Cathy', 'Doug'], dtype='<U10')

# Get first row of data
data[0]
('Alice', 25, 55.0)

# Get the name from the last row
data[-1]['name']
'Doug'
```

# Structured arrays

Using Boolean masking, this even allows you to do some more sophisticated operations such as filtering on age:

```
# Get names where age is under 30
data[data['age'] < 30]['name']
```

```
array(['Alice', 'Doug'], dtype='<U10')
```

# Creating structured arrays

the dictionary method:

```
np.dtype({'names':('name', 'age', 'weight'),
'formats':('U10', 'i4', 'f8')})
```

For clarity, numerical types can be specified using Python types or NumPy dtypes instead:

```
np.dtype({'names':('name', 'age', 'weight'),
'formats':((np.str_, 10), int, np.float32)})
```

A compound type can also be specified as a list of tuples:

```
np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
```

# Creating structured arrays

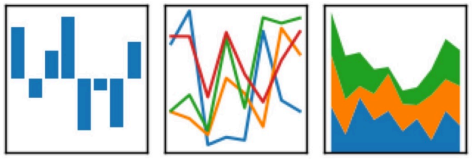
If the names of the types do not matter to you, you can specify the types alone in a comma-separated string:
np.dtype('S10,i4,f8')

| Character | Description | Example |
|---|---|---|
| 'b' | Byte | np.dtype('b') |
| 'i' | Signed integer | np.dtype('i4') == np.int32 |
| 'u' | Unsigned integer | np.dtype('u1') == np.uint8 |
| 'f' | Floating point | np.dtype('f8') == np.int64 |
| 'c' | Complex floating point | np.dtype('c16') == np.complex128 |
| 'S', 'a' | String | np.dtype('S5') |
| 'U' | Unicode string | np.dtype('U') == np.str_ |
| 'V' | Raw data (void) | np.dtype('V') == np.void |

# Pandas

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Luís Garmendia

# Introduction to Pandas

Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a DataFrame. DataFrames are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data.
As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

# Introduction to Pandas

NumPy's ndarray data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks. While it serves this purpose very well, its limitations become clear when we need more flexibility (e.g., attaching labels to data, working with missing data, etc.) and when attempting operations that do not map well to element-wise broadcasting (e.g., groupings, pivots, etc.)
Pandas, and in particular its **Series** and **DataFrame** objects, builds on the NumPy array structure

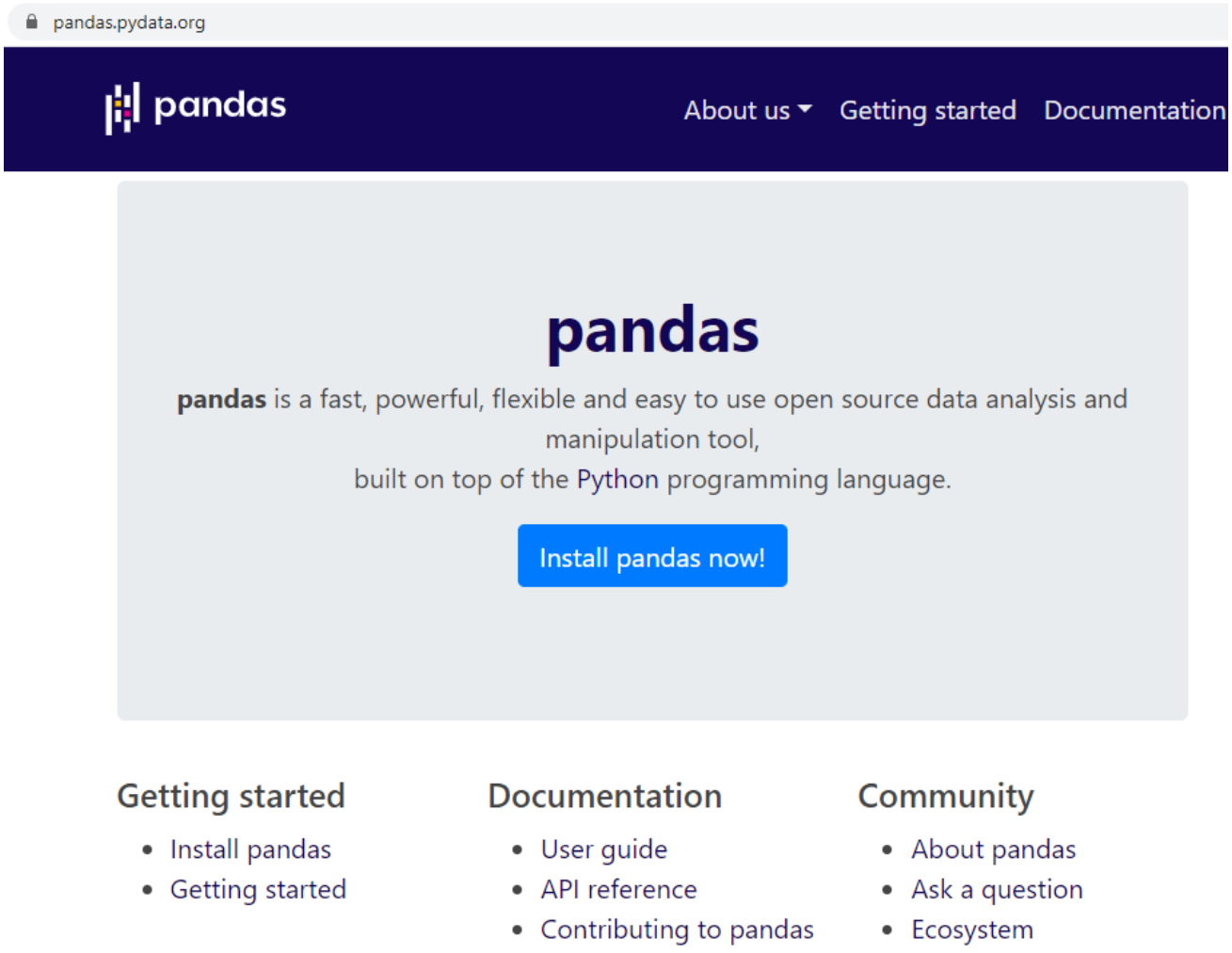
https://numpy.org/

# Installing Pandas
# pip install pandas

```
      1 import numpy as np
----> 2 import pandas as pd

ModuleNotFoundError: No module named 'pandas'

In [153]: pip install pandas
Collecting pandas
  Downloading pandas-1.1.5-cp37-cp37m-win32.whl (7.8 MB)
     |                                        | 7.8 MB 2.2 MB/s
Requirement already satisfied: numpy>=1.15.4 in c:\users\luis\appda
from pandas) (1.20.3)
Requirement already satisfied: python-dateutil>=2.7.3 in c:\users\l
ackages (from pandas) (2.8.1)
Collecting pytz>=2017.2
  Downloading pytz-2021.1-py2.py3-none-any.whl (510 kB)
     |                                        | 510 kB 6.4 MB/s
Requirement already satisfied: six>=1.5 in c:\users\luis\appdata\lo
python-dateutil>=2.7.3->pandas) (1.16.0)
Installing collected packages: pytz, pandas
Successfully installed pandas-1.1.5 pytz-2021.1
Note: you may need to restart the kernel to use updated packages.

In [154]: import pandas as pd
```

https://pandas.pydata.org/

# Installing Pandas




https://pandas.pydata.org/

# Installing Pandas

Create a new (Python 3) notebook

```
In [1]: import pandas

In [2]: pandas.__version__
Out[2]: '1.0.1'
```

https://pandas.pydata.org/

# Pandas objects

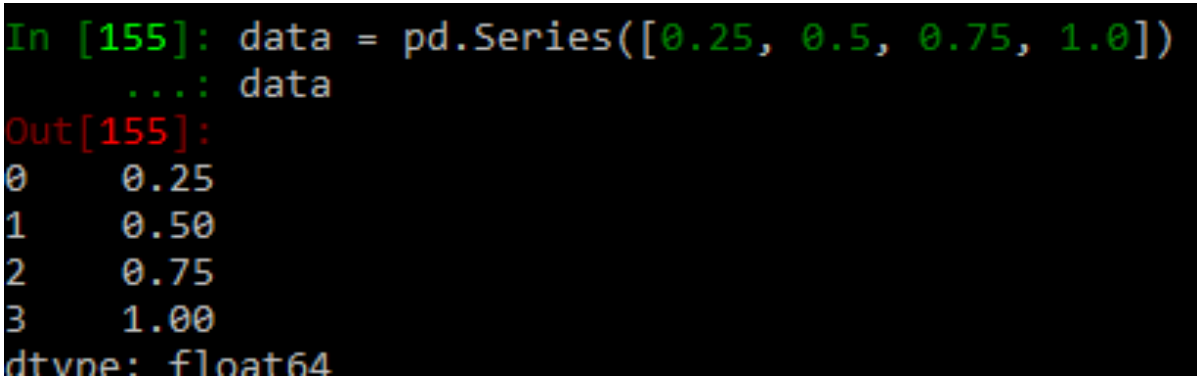
At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices

https://pandas.pydata.org/

# Pandas series objects

APandas Series is a one-dimensional array of indexed data. It can be created from a list or array

```
import numpy as np
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```




https://pandas.pydata.org/

# Pandas series objects

As we see in the output, the Series wraps both a sequence of values and a sequence of indices, which we can access with the values and index attributes.

```
data.values
data.index
data[1]
data[1:3]
```

https://pandas.pydata.org/

# series as generalized Numpy array

From what we've seen so far, it may look like the Series object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the Numpy Array has an *implicitly defined* integer index used to access the values, the Pandas Series has an ***explicitly defined*** index associated with the values.

This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type.

https://pandas.pydata.org/

# series as generalized Numpy array strings as an index

For example, if we wish, we can use strings as an index

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

```
data['b']
```

```
In [160]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
     ...:                  index=['a', 'b', 'c', 'd'])

In [161]: data
Out[161]:
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64

In [162]: data['b']
Out[162]: 0.5
```

https://pandas.pydata.org/

# series as generalized Numpy array non-sequential index

We can even use non-contiguous or non-sequential indices:

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

```
data[5]
```

```
In [163]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
     ...:                  index=[2, 5, 3, 7])
     ...: data
Out[163]:
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64

In [164]: data[5]
Out[164]: 0.5
```

https://pandas.pydata.org/

# series as specialized dictionary

You can think of a Pandas Series a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
population
```

```
Out[165]:
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

https://pandas.pydata.org/

# Constructing series

pd.Series(data, index=index)
where index is an optional argument

```
pd.Series([2, 4, 6])
pd.Series(5, index=[100, 200, 300])
pd.Series({2:'a', 1:'b', 3:'c'})
pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```

https://pandas.pydata.org/

# Constructing series

```
In [166]: pd.Series([2, 4, 6])
Out[166]:
0    2
1    4
2    6
dtype: int64

In [167]: pd.Series(5, index=[100, 200, 300])
Out[167]:
100    5
200    5
300    5
dtype: int64

In [168]: pd.Series({2:'a', 1:'b', 3:'c'})
Out[168]:
2    a
1    b
3    c
dtype: object

In [169]: pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
Out[169]:
3    c
2    a
dtype: object
```

# Pandas DataFrame object

the DataFrame can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary

If a Series is an analog of a one-dimensional array with flexible indices, a DataFrame is an analog of a two-dimensional array with both flexible row indices and flexible column names

https://pandas.pydata.org/

# Pandas DataFrame object

```
area_dict = {'California': 423967,
             'Texas': 695662,
             'New York': 141297,
             'Florida': 170312,
             'Illinois': 149995}
area = pd.Series(area_dict)
area
```

```
Out[165]:
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

https://pandas.pydata.org/

# Pandas DataFrame object

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

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

https://pandas.pydata.org/

# Pandas DataFrame object

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

States

```
     ...: population
Out[173]:
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64

In [174]: area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
     ...:                'Florida': 170312, 'Illinois': 149995}
     ...: area = pd.Series(area_dict)
     ...: area
Out[174]:
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
dtype: int64

In [175]: states = pd.DataFrame({'population': population,
     ...:                        'area': area})
     ...: states
Out[175]:
            population    area
California    38332521  423967
Texas         26448193  695662
New York      19651127  141297
Florida       19552860  170312
Illinois      12882135  149995
```

# Pandas DataFrame object

states.index

states.columns

```
In [176]: states.index
Out[176]: Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')

In [177]: states.columns
Out[177]: Index(['population', 'area'], dtype='object')
```

https://pandas.pydata.org/

# DataFrame as a specialized dictionary

For a DataFrame, data['col0'] will return the first *column*. Because of this, it is probably better to think about DataFrames as **generalized dictionaries** rather than generalized arrays, though both ways of looking at the situation can be useful.

states['area']

```
In [180]: states['area']
Out[180]:
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

# Constructing DataFrame objects From Series objects

A DataFrame is a collection of Series objects, and a single-column DataFrame can be constructed from a single Series

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

```
In [183]: pd.DataFrame(population, columns=['population'])
Out[183]:
            population
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
```

# Constructing DataFrame objects From lists of dicts

Any list of dictionaries can be made into a DataFrame

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

```
In [184]: data = [{'a': i, 'b': 2 * i}
     ...:               for i in range(3)]
     ...: pd.DataFrame(data)
Out[184]:
   a  b
0  0  0
1  1  2
2  2  4
```

# Constructing DataFrame objects From lists of dicts

Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e., "not a number") values:
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])

```
In [185]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
Out[185]:
     a  b    c
0  1.0  2  NaN
1  NaN  3  4.0
```

https://pandas.pydata.org/

# Constructing DataFrame objects From dictionary of series objects

A DataFrame can be constructed from a dictionary of Series

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

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

# Constructing DataFrame objects From two-dimensional Numpy arrays

Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names. If omitted, an integer index will be used for each

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

```
In [186]: pd.DataFrame(np.random.rand(3, 2),
     ...:                columns=['foo', 'bar'],
     ...:                index=['a', 'b', 'c'])
Out[186]:
        foo       bar
a  0.317292  0.373789
b  0.749909  0.457733
c  0.679485  0.112729
```

# Constructing DataFrame objects From a NumPy structured array

Pandas DataFrame operates much like a structured array, and can be created directly from one

```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
A
pd.DataFrame(A)
```

```
In [187]: A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
     ...: A
     ...: pd.DataFrame(A)
     ...:
Out[187]:
   A    B
0  0  0.0
1  0  0.0
2  0  0.0
```

# Pandas index object

We have seen here that both
the Series and DataFrame objects contain an
explicit *index* that lets you reference and modify data.

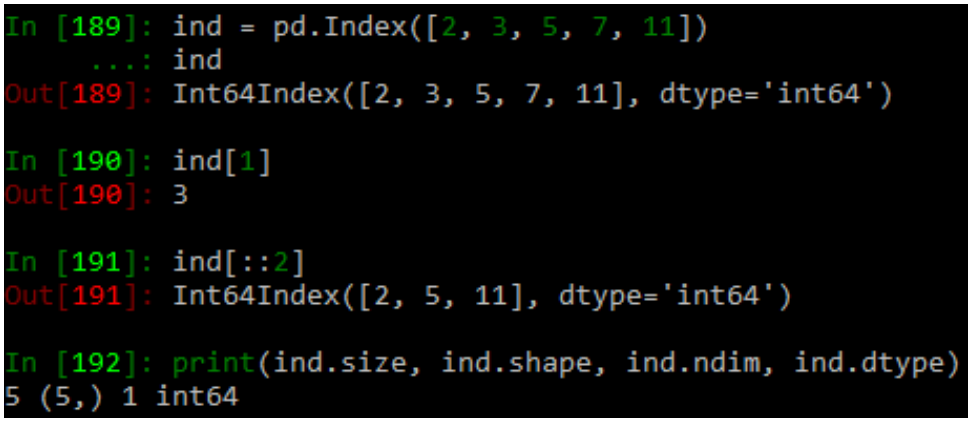
This Index object is an interesting structure in itself, and it can be thought of either as an *immutable array* or as an *ordered set* (technically a multi-set, as Index objects may contain repeated values).

https://pandas.pydata.org/

# Pandas index object as immutable array

The Index in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices

```
ind = pd.Index([2, 3, 5, 7, 11])
Ind


ind[1]


ind[::2]
print(ind.size, ind.shape, ind.ndim, ind.dtype)
```

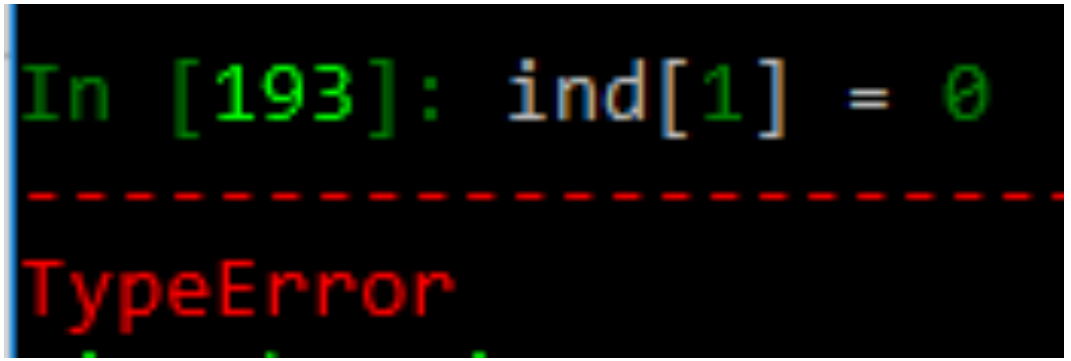


# Pandas index object as immutable array

One difference between Index objects and NumPy arrays is that indices are immutable–that is, they cannot be modified

This immutability makes it safer to share indices between multiple DataFrames and arrays, without the potential for side effects from inadvertent index modification.

ind[1] = 0



# Pandas index object as ordered set

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. The Index object follows many of the conventions used by Python's built-in set data structure, so that unions, intersections, differences, and other combinations

```
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
```

```
In [195]: indA = pd.Index([1, 3, 5, 7, 9])
     ...: indB = pd.Index([2, 3, 5, 7, 11])

In [196]: indA & indB
Out[196]: Int64Index([3, 5, 7], dtype='int64')

In [197]: indA | indB  # union
Out[197]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

In [198]: indA ^ indB  # symmetric difference
Out[198]: Int64Index([1, 2, 9, 11], dtype='int64')
```

# Ufuncs on Series and DataFrame

Because Pandas is designed to work with NumPy, any NumPy ufunc will work on Series and DataFrame objects

```
import pandas as pd
import numpy as np
rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
df = pd.DataFrame(rng.randint(0, 10, (3, 4))
          , columns=['A', 'B', 'C', 'D'])
np.exp(ser)
np.sin(df * np.pi / 4)
```

# Ufuncs on Series and DataFrame

```
In [205]: rng = np.random.RandomState(42)

In [206]: ser = pd.Series(rng.randint(0, 10, 4))
     ...: ser
Out[206]:
0    6
1    3
2    7
3    4
dtype: int32

In [207]: df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
     ...:                           columns=['A', 'B', 'C', 'D'])
     ...: df
Out[207]:
   A  B  C  D
0  6  9  2  6
1  7  4  3  7
2  7  2  5  4

In [208]: np.exp(ser)
Out[208]:
0     403.428793
1      20.085537
2    1096.633158
3      54.598150
dtype: float64

In [209]: np.sin(df * np.pi / 4)
Out[209]:
          A             B         C             D
0 -1.000000  7.071068e-01  1.000000 -1.000000e+00
1 -0.707107  1.224647e-16  0.707107 -7.071068e-01
2 -0.707107  1.000000e+00 -0.707107  1.224647e-16
```

# Index alignments in Series

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas':
26448193, 'New York': 19651127}, name='population')
population / area
```

```
In [210]: area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
     ...:                   'California': 423967}, name='area')
     ...: population = pd.Series({'California': 38332521, 'Texas': 26448193,
     ...:                         'New York': 19651127}, name='population')

In [211]: population / area
Out[211]:
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

# Index alignments in Series fill_value

If using NaN values is not the desired behavior, the fill value can be used:

```
A = pd.Series([2, 4, 6],
    index=[0, 1, 2])
B = pd.Series([1, 3, 5],
    index=[1, 2, 3])
A + B
```

```
A.add(B, fill_value=0)
```

```
In [215]: A
Out[215]:
0    2
1    4
2    6
dtype: int64

In [216]: B = pd.Series([1, 3, 5], index=[1, 2, 3])

In [217]: B
Out[217]:
1    1
2    3
3    5
dtype: int64

In [218]: A+B
Out[218]:
0    NaN
1    5.0
2    9.0
3    NaN
dtype: float64

In [219]: A.add(B, fill_value=0)
Out[219]:
0    2.0
1    5.0
2    9.0
3    5.0
dtype: float64
```

# Index alignments in DataFrames

A similar type of alignment takes place for *both* columns and indices when performing operations on DataFrame

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
    columns=list('AB'))
A
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
columns=list('BAC'))
B
A + B
fill = A.stack().mean()
A.add(B, fill_value=fill)
```

# Index alignments in DataFrames

```
In [220]: A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
     ...:                  columns=list('AB'))
     ...: A
Out[220]:
   A   B
0  1  11
1  5   1

In [221]: B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
     ...:                  columns=list('BAC'))
     ...: B
Out[221]:
   B  A  C
0  4  0  9
1  5  8  0
2  9  2  6

In [222]: A + B
Out[222]:
      A     B   C
0   1.0  15.0 NaN
1  13.0   6.0 NaN
2   NaN   NaN NaN

In [223]: fill = A.stack().mean()
     ...: A.add(B, fill_value=fill)
Out[223]:
      A     B     C
0   1.0  15.0  13.5
1  13.0   6.0   4.5
2   6.5  13.5  10.5

In [224]: fill
Out[224]: 4.5
```

# Ufuncs with Data Frames and Series

When performing operations between a DataFrame and a Series, the index and column alignment is similarly maintained.

```
A = rng.randint(10, size=(3, 4))
A
```

```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]  # substract row 0
```

```
df.subtract(df['R'], axis=0)  #substract column R
```

# Ufuncs with Data Frames and Series

```
In [225]: A = rng.randint(10, size=(3, 4))
     ...: A
Out[225]:
array([[3, 8, 2, 4],
       [2, 6, 4, 8],
       [6, 1, 3, 8]])

In [226]: df = pd.DataFrame(A, columns=list('QRST'))

In [227]: df
Out[227]:
   Q  R  S  T
0  3  8  2  4
1  2  6  4  8
2  6  1  3  8

In [228]: df.iloc[0]
Out[228]:
Q    3
R    8
S    2
T    4
Name: 0, dtype: int32

In [229]: df - df.iloc[0]
Out[229]:
   Q  R  S  T
0  0  0  0  0
1 -1 -2  2  4
2  3 -7  1  4

In [230]: df.subtract(df['R'], axis=0)
Out[230]:
   Q  R  S  T
0 -5  0 -6 -4
1 -4  0 -2  2
```

# Ufuncs with Data Frames and Series

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

# Missing data in Pandas
# None

The first sentinel value used by Pandas is None, a Python singleton object that is often used for missing data in Python code. Because it is a Python object, None cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' (i.e., arrays of Python objects)

```
import numpy as np
import pandas as pd
vals1 = np.array([1, None, 3, 4])
vals1.sum() TypeError
```

# Missing data in Pandas
# NaN

The other missing data representation, NaN (acronym for *Not a Number*), is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation

```
vals2 = np.array([1, np.nan, 3, 4])
0 * np.nan
1 + np.nan
vals2.sum(), vals2.min(), vals2.max()
```

```
In [231]: vals2 = np.array([1, np.nan, 3, 4])
     ...: vals2.dtype
Out[231]: dtype('float64')

In [232]: 1 + np.nan
Out[232]: nan

In [233]: 0 *  np.nan
Out[233]: nan

In [234]: vals2.sum(), vals2.min(), vals2.max()
Out[234]: (nan, nan, nan)
```

# Missing data in Pandas
# NaN

NumPy does provide some special aggregations that will ignore these missing values:

np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)

```
In [236]: vals2
Out[236]: array([ 1., nan,  3.,  4.])

In [237]: np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
Out[237]: (8.0, 1.0, 4.0)
```

# Missing data in Pandas
# NaN

- isnull(): Generate a boolean mask indicating missing values
- notnull(): Opposite of isnull()
- dropna(): Return a filtered version of the data
- fillna(): Return a copy of the data with missing values filled or imputed

# Concat and Append

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges that correctly handle any overlaps between the datasets.

# Concat and Append
# pd.concat series

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

```
In [238]: ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
     ...: ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
     ...: pd.concat([ser1, ser2])
Out[238]:
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

# Concat and Append
# pd.concat DataFrames

```
df1     df2     pd.concat([df1, df2])
```

| | A | B |
|---|---|---|
| 1 | A1 | B1 |
| 2 | A2 | B2 |

| | A | B |
|---|---|---|
| 3 | A3 | B3 |
| 4 | A4 | B4 |

| | A | B |
|---|---|---|
| 1 | A1 | B1 |
| 2 | A2 | B2 |
| 3 | A3 | B3 |
| 4 | A4 | B4 |

# Concat and Append pd.concat DataFrames

```
df3     df4     pd.concat([df3, df4], axis='col')
```

| | A | B |
|---|---|---|
| 0 | A0 | B0 |
| 1 | A1 | B1 |

| | C | D |
|---|---|---|
| 0 | C0 | D0 |
| 1 | C1 | D1 |

| | A | B | C | D |
|---|---|---|---|---|
| 0 | A0 | B0 | C0 | D0 |
| 1 | A1 | B1 | C1 | D1 |

# Concat and Append pd.concat DataFrames ignoring indices

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

| | A | B |
|---|---|---|
| 0 | A0 | B0 |
| 1 | A1 | B1 |

| | A | B |
|---|---|---|
| 0 | A2 | B2 |
| 1 | A3 | B3 |

| | A | B |
|---|---|---|
| 0 | A0 | B0 |
| 1 | A1 | B1 |
| 2 | A2 | B2 |
| 3 | A3 | B3 |

# Concat and Append pd.concat DataFrames adding MultiIndex keys

```
x       y       pd.concat([x, y], keys=['x', 'y'])
```

| | A | B |
|---|---|---|
| 0 | A0 | B0 |
| 1 | A1 | B1 |

| | A | B |
|---|---|---|
| 0 | A2 | B2 |
| 1 | A3 | B3 |

| | | A | B |
|---|---|---|---|
| x | 0 | A0 | B0 |
| | 1 | A1 | B1 |
| y | 0 | A2 | B2 |
| | 1 | A3 | B3 |

# Concat and Append pd.concat DataFrames joins

```
df5      df6      pd.concat([df5, df6], join='inner')
```

| | A | B | C |
|---|---|---|---|
| 1 | A1 | B1 | C1 |
| 2 | A2 | B2 | C2 |

| | B | C | D |
|---|---|---|---|
| 3 | B3 | C3 | D3 |
| 4 | B4 | C4 | D4 |

| | B | C |
|---|---|---|
| 1 | B1 | C1 |
| 2 | B2 | C2 |
| 3 | B3 | C3 |
| 4 | B4 | C4 |

# Append

Because direct array concatenation is so common, Series and DataFrame objects have an append method that can accomplish the same thing in fewer keystrokes.

The append() method in Pandas does not modify the original object–instead it creates a new object with the combined data.

# Append

`df1` `df2` `df1.append(df2)`

| | A | B |
|---|---|---|
| 1 | A1 | B1 |
| 2 | A2 | B2 |

| | A | B |
|---|---|---|
| 3 | A3 | B3 |
| 4 | A4 | B4 |

| | A | B |
|---|---|---|
| 1 | A1 | B1 |
| 2 | A2 | B2 |
| 3 | A3 | B3 |
| 4 | A4 | B4 |

# Merge
# One-to-one join

The pd.merge() function implements a number of types of joins: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the pd.merge() interface; the type of join performed depends on the form of the input data.

# Merge
# One-to-one join

```
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]}) display('df1', 'df2')

df3 = pd.merge(df1, df2)
df3
```

# Merge
# One-to-one join

`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 |

To combine this information into a single `DataFrame`, we can use the `pd.merge()` function:

```
df3 = pd.merge(df1, df2)
df3
```

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

# Merge
# Many-to-one join

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

df3

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

df4

| | group | supervisor |
|---|---|---|
| 0 | Accounting | Carly |
| 1 | Engineering | Guido |
| 2 | HR | Steve |

pd.merge(df3, df4)

| | employee | group | hire_date | supervisor |
|---|---|---|---|---|
| 0 | Bob | Accounting | 2008 | Carly |
| 1 | Jake | Engineering | 2012 | Guido |
| 2 | Lisa | Engineering | 2004 | Guido |
| 3 | Sue | HR | 2014 | Steve |

# Merge
# Many-to-many join

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

# Merge
# Many-to-many join

`df1`

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

`df5`

| | group | skills |
|---|---|---|
| 0 | Accounting | math |
| 1 | Accounting | spreadsheets |
| 2 | Engineering | coding |
| 3 | Engineering | linux |
| 4 | HR | spreadsheets |
| 5 | HR | organization |

```
pd.merge(df1, df5)
```

| | employee | group | skills |
|---|---|---|---|
| 0 | Bob | Accounting | math |
| 1 | Bob | Accounting | spreadsheets |
| 2 | Jake | Engineering | coding |
| 3 | Jake | Engineering | linux |
| 4 | Lisa | Engineering | coding |
| 5 | Lisa | Engineering | linux |
| 6 | Sue | HR | spreadsheets |
| 7 | Sue | HR | organization |

# Merge
# Merge key

display('df1', 'df2', "pd.merge(df1, df2, on='employee')")

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 |

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

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

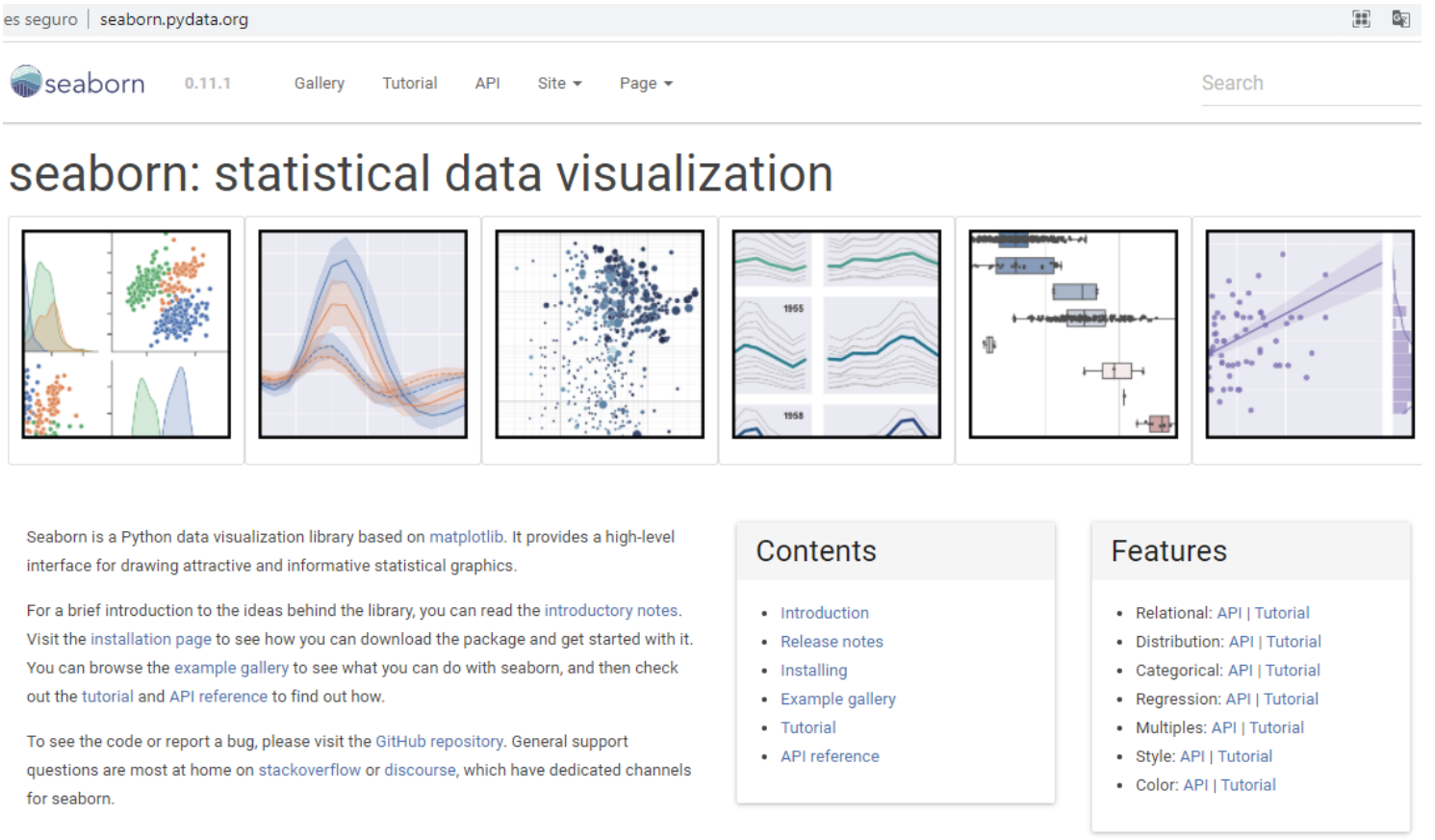
# Aggregation and groupping

An essential piece of analysis of large data is efficient summarization: computing aggregations
like sum(), mean(), median(), min(), and max(), in which a single number gives insight into the nature of a potentially large dataset.

This can be extended to more sophisticated operations based on the concept of a groupby.

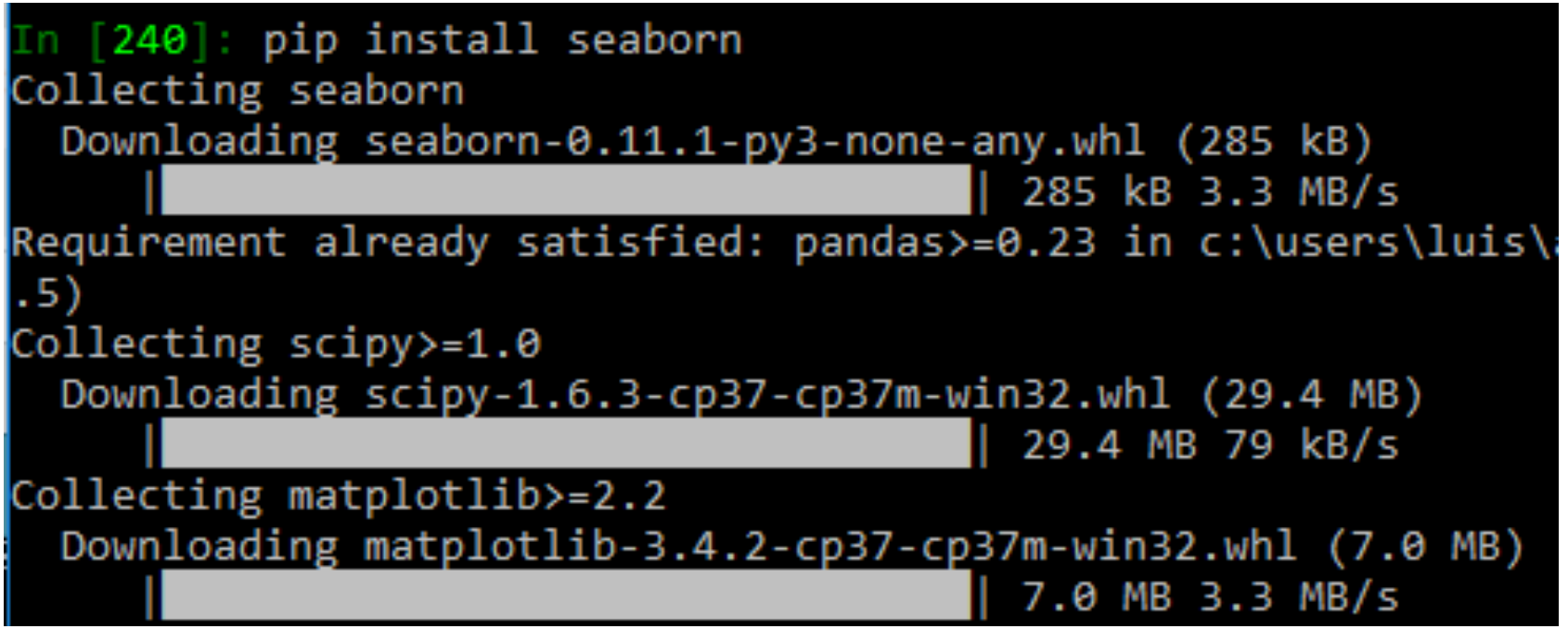
# Seaborn package

http://seaborn.pydata.org/



# Install seaborn
# pip install seaborn

http://seaborn.pydata.org/



# Planets data

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
planets.head()
```

```
In [1]: import seaborn as sns
   ...: planets = sns.load_dataset('planets')
   ...: planets.shape
Matplotlib is building the font cache; this may take a moment.
Out[1]: (1035, 6)

In [2]: planets.head()
Out[2]:
            method  number  orbital_period   mass  distance  year
0  Radial Velocity       1         269.300   7.10     77.40  2006
1  Radial Velocity       1         874.774   2.21     56.95  2008
2  Radial Velocity       1         763.000   2.60     19.84  2011
3  Radial Velocity       1         326.030  19.40    110.62  2007
4  Radial Velocity       1         516.220  10.50    119.47  2009
```

# .describe

planets.dropna().describe()

```
In [3]: planets.dropna().describe()
Out[3]:
          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
```

# .describe

The following table summarizes some other built-in Pandas aggregations:

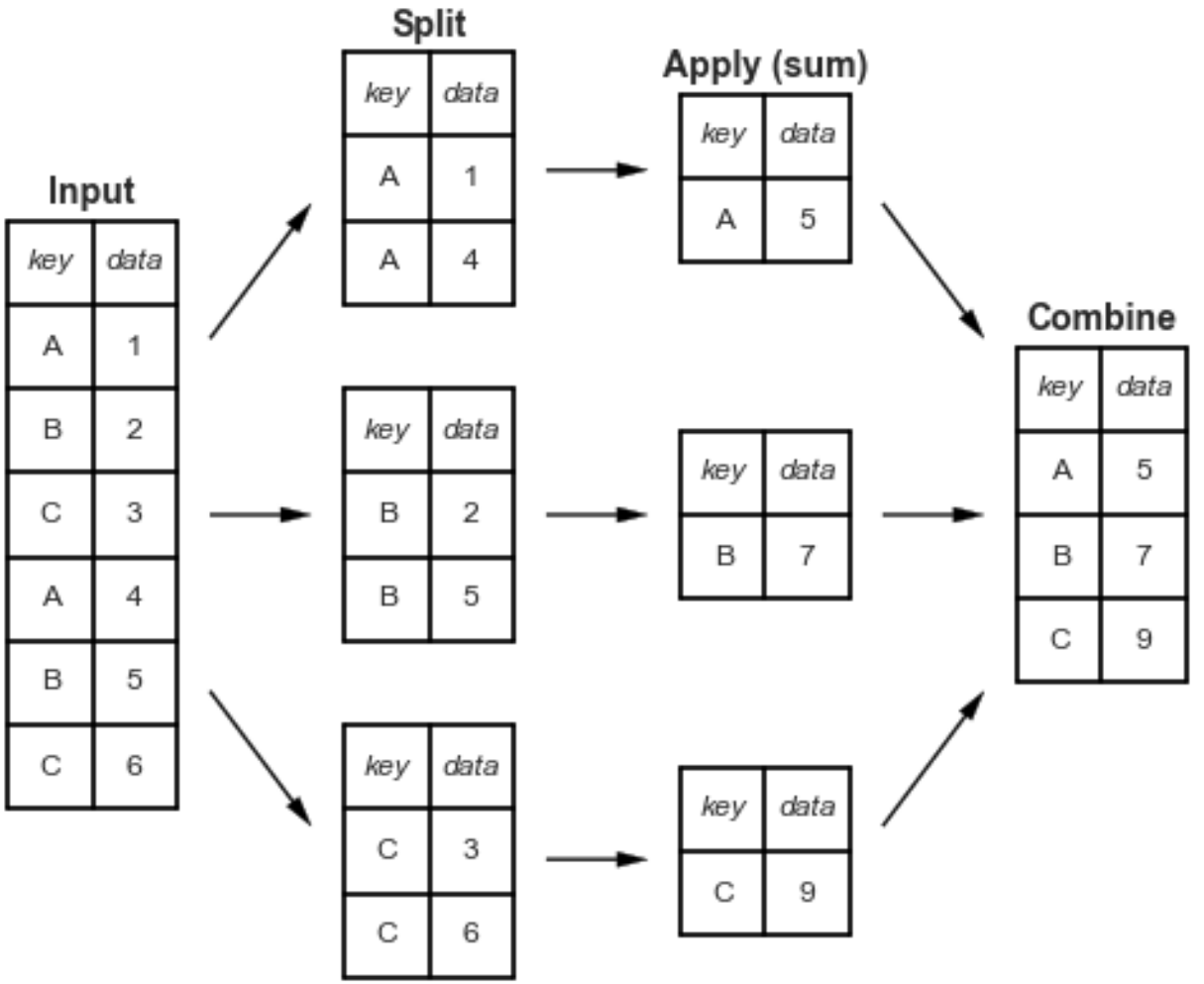
| Aggregation | Description |
| --- | --- |
| count() | Total number of items |
| first(), last() | First and last item |
| mean(), median() | Mean and median |
| min(), max() | Minimum and maximum |
| std(), var() | Standard deviation and variance |
| mad() | Mean absolute deviation |
| prod() | Product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

# GroupBy: Split, Apply, Combine

We can aggregate conditionally on some label or index: this is implemented in the so-called groupby operation
This makes clear what the groupby accomplishes:

- The ***split*** step involves breaking up and grouping a DataFrame depending on the value of the specified key.
- The ***apply*** step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
- The ***combine*** step merges the results of these operations into an output array.

# GroupBy: Split, Apply, Combine



# GroupBy
# Columns indexing

The GroupBy object supports column indexing in the same way as the DataFrame, and returns a modified GroupBy object.

```
planets.groupby('method')
planets.groupby('method')['orbital_period']
planets.groupby('method')['orbital_period'].median()
```

```
In [4]: planets.groupby('method')
Out[4]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x19C57E30>

In [5]: planets.groupby('method')['orbital_period']
Out[5]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x19C57370>

In [6]: planets.groupby('method')['orbital_period'].median()
Out[6]:
method
Astrometry                         631.180000
Eclipse Timing Variations         4343.500000
Imaging                          27500.000000
Microlensing                      3300.000000
Orbital Brightness Modulation        0.342887
Pulsar Timing                       66.541900
Pulsation Timing Variations       1170.000000
Radial Velocity                    360.200000
Transit                              5.714932
Transit Timing Variations           57.011000
Name: orbital_period, dtype: float64
```

# Pivot Tables

The pivot table takes simple column-wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data

Think of pivot tables as essentially
a *multidimensional* version of GroupBy aggregation

# Passengers of the Titanic

```
import numpy as np
import pandas as pd
import seaborn as sns
titanic = sns.load_dataset('titanic')
titanic.head()
```

```
In [7]: import numpy as np
   ...: import pandas as pd
   ...: import seaborn as sns
   ...: titanic = sns.load_dataset('titanic')

In [8]: titanic.head()
Out[8]:
   survived  pclass     sex   age  sibsp  parch     fare  ...  class    who adult_male  deck  embark_town alive  alone
0         0       3    male  22.0      1      0   7.2500  ...  Third    man       True   NaN  Southampton    no  False
1         1       1  female  38.0      1      0  71.2833  ...  First  woman      False     C    Cherbourg   yes  False
2         1       3  female  26.0      0      0   7.9250  ...  Third  woman      False   NaN  Southampton   yes   True
3         1       1  female  35.0      1      0  53.1000  ...  First  woman      False     C  Southampton   yes  False
4         0       3    male  35.0      0      0   8.0500  ...  Third    man       True   NaN  Southampton    no   True

[5 rows x 15 columns]
```

# Pivot tables by hand

titanic.groupby('sex')[['survived']].mean()

```
In [9]: titanic.groupby('sex')[['survived']].mean()
Out[9]:
        survived
sex
female  0.742038
male    0.188908
```

# Pivot tables by hand unstack

titanic.groupby(['sex', 'class']) ['survived'].aggregate('mean').unstack()

```
In [10]: titanic.groupby(['sex', 'class'])['survived'].aggregate('mean')
Out[10]:
sex     class
female  First     0.968085
        Second    0.921053
        Third     0.500000
male    First     0.368852
        Second    0.157407
        Third     0.135447
Name: survived, dtype: float64

In [11]: titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
Out[11]:
class      First    Second     Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

# Pivot tables syntax

Here is the equivalent to the preceding operation using the pivot_table method of DataFrames

titanic.pivot_table('survived', index='sex', columns='class')

```
In [12]: titanic.pivot_table('survived', index='sex', columns='class')
Out[12]:
class      First    Second     Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

# Multi-level Pivot tables

Just as in the GroupBy, the grouping in pivot tables can be specified with multiple levels. We'll bin the age using the pd.cut function

```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', age], 'class')
```

```
In [13]: age = pd.cut(titanic['age'], [0, 18, 80])^M
    ...: titanic.pivot_table('survived', ['sex', age], 'class')
Out[13]:
class                  First    Second     Third
sex    age
female (0, 18]      0.909091  1.000000  0.511628
       (18, 80]     0.972973  0.900000  0.423729
male   (0, 18]      0.800000  0.600000  0.215686
       (18, 80]     0.375000  0.071429  0.133663
```

# Multi-level Pivot tables

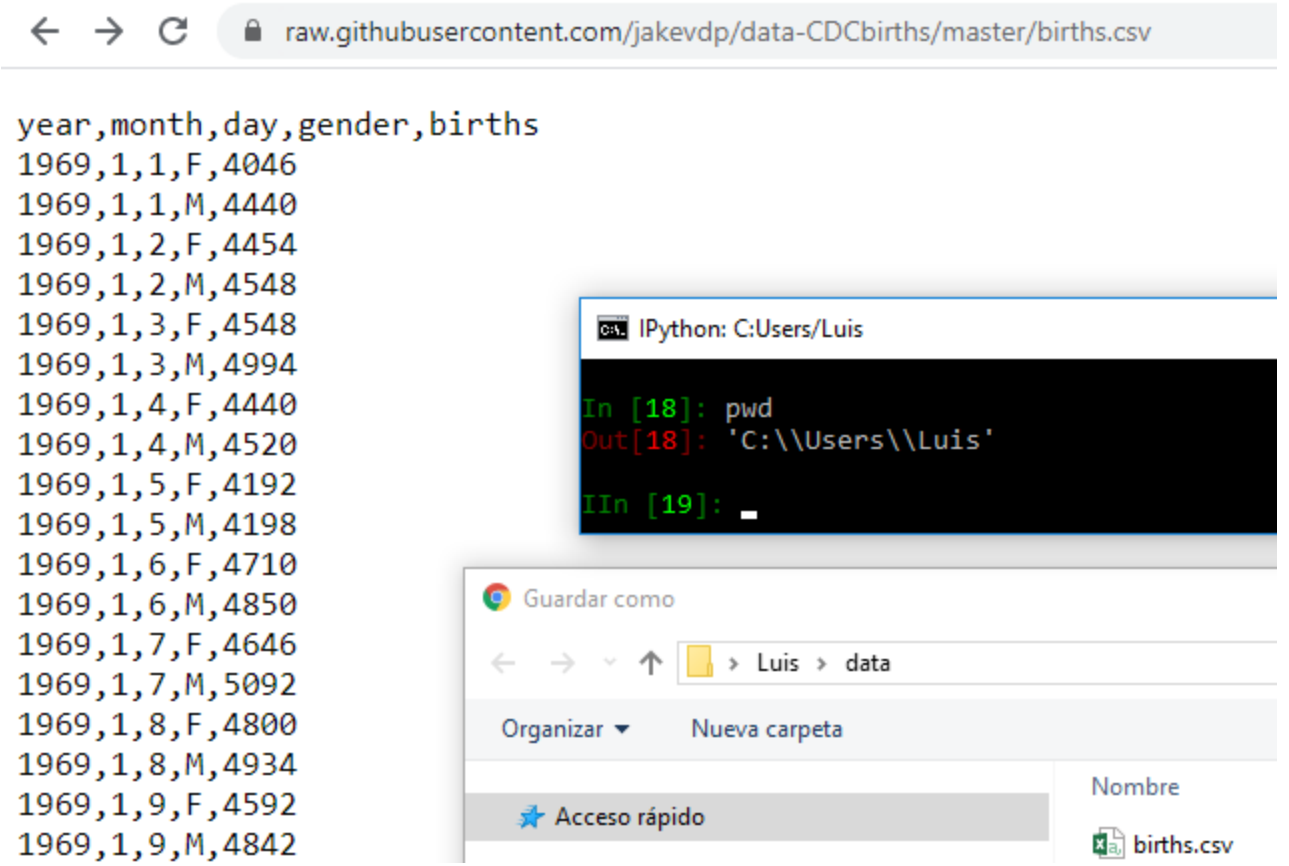
We can apply the same strategy when working with the columns as well; let's add info on the fare paid using pd.qcut to automatically compute quantiles

```
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

```
In [14]: fare = pd.qcut(titanic['fare'], 2)^M
    ...: titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
Out[14]:
fare            (-0.001, 14.454]                    (14.454, 512.329]
class                      First    Second     Third             First    Second     Third
sex    age
female (0, 18]               NaN  1.000000  0.714286          0.909091  1.000000  0.318182
       (18, 80]              NaN  0.880000  0.444444          0.972973  0.914286  0.391304
male   (0, 18]               NaN  0.000000  0.260870          0.800000  0.818182  0.178571
       (18, 80]              0.0  0.098039  0.125000          0.391304  0.030303  0.192308
```

# Download births.csv

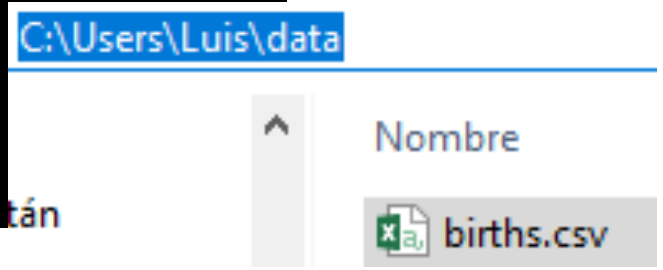
https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv



# Download births.csv

https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv

```
In [18]: pwd
Out[18]: 'C:\\Users\\Luis'

In [19]: births = pd.read_csv('data/births.csv')

In [20]: births.head()
Out[20]:
   year  month  day gender  births
0  1969      1  1.0      F    4046
1  1969      1  1.0      M    4440
2  1969      1  2.0      F    4454
3  1969      1  2.0      M    4548
4  1969      1  3.0      F    4548
```



# Download births.csv jupyter notebook

!curl -O https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv

In [7]:
```
!curl -O https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100  258k  100  258k    0     0  2247k      0 --:--:-- --:--:-- --:--:-- 2267k
```

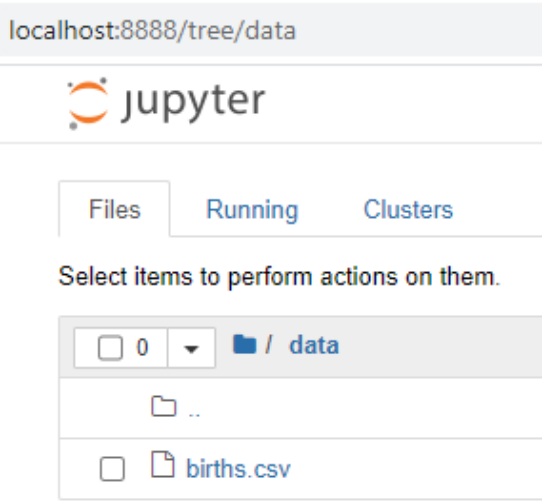
In [8]:
```
import pandas as pd
```

In [9]:
```
births = pd.read_csv('data/births.csv')
```

In [10]:
```
births.head()
```

Out[10]:

| | year | month | day | gender | births |
|---|---|---|---|---|---|
| 0 | 1969 | 1 | 1.0 | F | 4046 |
| 1 | 1969 | 1 | 1.0 | M | 4440 |
| 2 | 1969 | 1 | 2.0 | F | 4454 |
| 3 | 1969 | 1 | 2.0 | M | 4548 |
| 4 | 1969 | 1 | 3.0 | F | 4548 |



# pivot

['decade'] = 10 * (births['year'] // 10) births.pivot_table('births', index='decade', columns='gender', aggfunc='sum')

```
In [4]: births['decade'] = 10 * (births['year'] // 10)
        births.pivot_table('births', index='decade', columns='gender', aggfunc='sum'
```

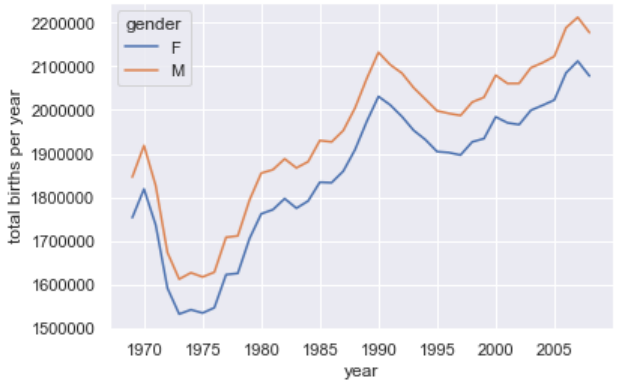
Out[4]:

| gender | F | M |
|---|---|---|
| decade | | |
| 1960 | 1753634 | 1846572 |
| 1970 | 16263075 | 17121550 |
| 1980 | 18310351 | 19243452 |
| 1990 | 19479454 | 20420553 |
| 2000 | 18229309 | 19106428 |

# Pivot.plot

```
%matplotlib inline
import matplotlib.pyplot as plt
sns.set() # use Seaborn styles
births.pivot_table('births', index='year', columns='gender',
aggfunc='sum').plot()
plt.ylabel('total births per year');
```

```
In [5]: import seaborn as sns

In [6]: %matplotlib inline
        import matplotlib.pyplot as plt
        sns.set()  # use Seaborn styles
        births.pivot_table('births', index='year', columns='gender', aggfunc='sum').plot()
        plt.ylabel('total births per year');
```



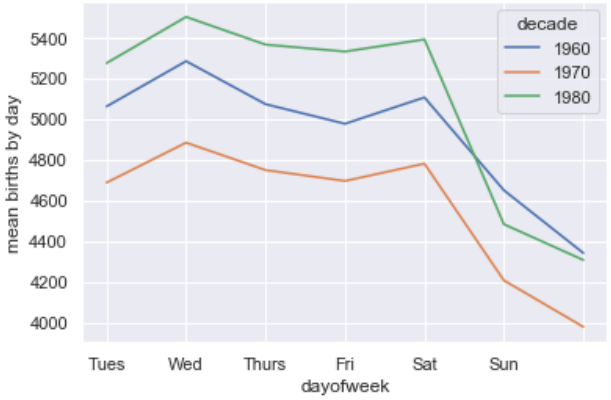
# Pivot

```
quartiles = np.percentile(births['births'], [25, 50, 75])
mu = quartiles[1]
sig = 0.74 * (quartiles[2] - quartiles[0])

births = births.query('(births > @mu - 5 * @sig) & (births < @mu
+ 5 * @sig)')

# set 'day' column to integer; it originally was a string due to nulls
births['day'] = births['day'].astype(int)

# create a datetime index from the year, month, day
births.index = pd.to_datetime(10000 * births.year + 100 *
    births.month + births.day, format='%Y%m%d')
births['dayofweek'] = births.index.dayofweek
```

# Pivot.plot

```
import matplotlib.pyplot as plt
import matplotlib as mpl
births.pivot_table('births', index='dayofweek',
        columns='decade', aggfunc='mean').plot()
plt.gca().set_xticklabels(['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'])
plt.ylabel('mean births by day');
```

```
In [14]: import matplotlib.pyplot as plt
         import matplotlib as mpl

         births.pivot_table('births', index='dayofweek',
                            columns='decade', aggfunc='mean').plot()
         plt.gca().set_xticklabels(['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'])
         plt.ylabel('mean births by day');
```



# Pivot

```
births_by_date = births.pivot_table('births', [births.index.month, births.index.day])
births_by_date.head()
births_by_date.index = [pd.datetime(2012, month, day)
                        for (month, day) in  births_by_date.index]
births_by_date.head()
```

In [15]:
```
births_by_date = births.pivot_table('births',
                                    [births.index.month, births.index.day])
births_by_date.head()
```

Out[15]:

| | | births |
|---|---|---|
| 1 | 1 | 4009.225 |
| | 2 | 4247.400 |
| | 3 | 4500.900 |
| | 4 | 4571.350 |
| | 5 | 4603.625 |

In [16]:
```
births_by_date.index = [pd.datetime(2012, month, day)
                        for (month, day) in births_by_date.index]
births_by_date.head()
```

```
C:\Users\Luis\anaconda3\lib\site-packages\ipykernel_launcher.py:2: FutureWar
ill be removed from pandas in a future version. Import from datetime module
```

Out[16]:

| | births |
|---|---|
| 2012-01-01 | 4009.225 |
| 2012-01-02 | 4247.400 |
| 2012-01-03 | 4500.900 |
| 2012-01-04 | 4571.350 |
| 2012-01-05 | 4603.625 |

# Pivot

```
births_by_date = births.pivot_table('births', [births.index.month, births.index.day])
births_by_date.head()
births_by_date.index = [pd.datetime(2012, month, day)
                        for (month, day) in  births_by_date.index]
births_by_date.head()
```

In [15]:
```
births_by_date = births.pivot_table('births',
                                    [births.index.month, births.index.day])
births_by_date.head()
```

Out[15]:

| | | births |
|---|---|---|
| 1 | 1 | 4009.225 |
| | 2 | 4247.400 |
| | 3 | 4500.900 |
| | 4 | 4571.350 |
| | 5 | 4603.625 |

In [16]:
```
births_by_date.index = [pd.datetime(2012, month, day)
                        for (month, day) in births_by_date.index]
births_by_date.head()
```

```
C:\Users\Luis\anaconda3\lib\site-packages\ipykernel_launcher.py:2: FutureWar
ill be removed from pandas in a future version. Import from datetime module
```

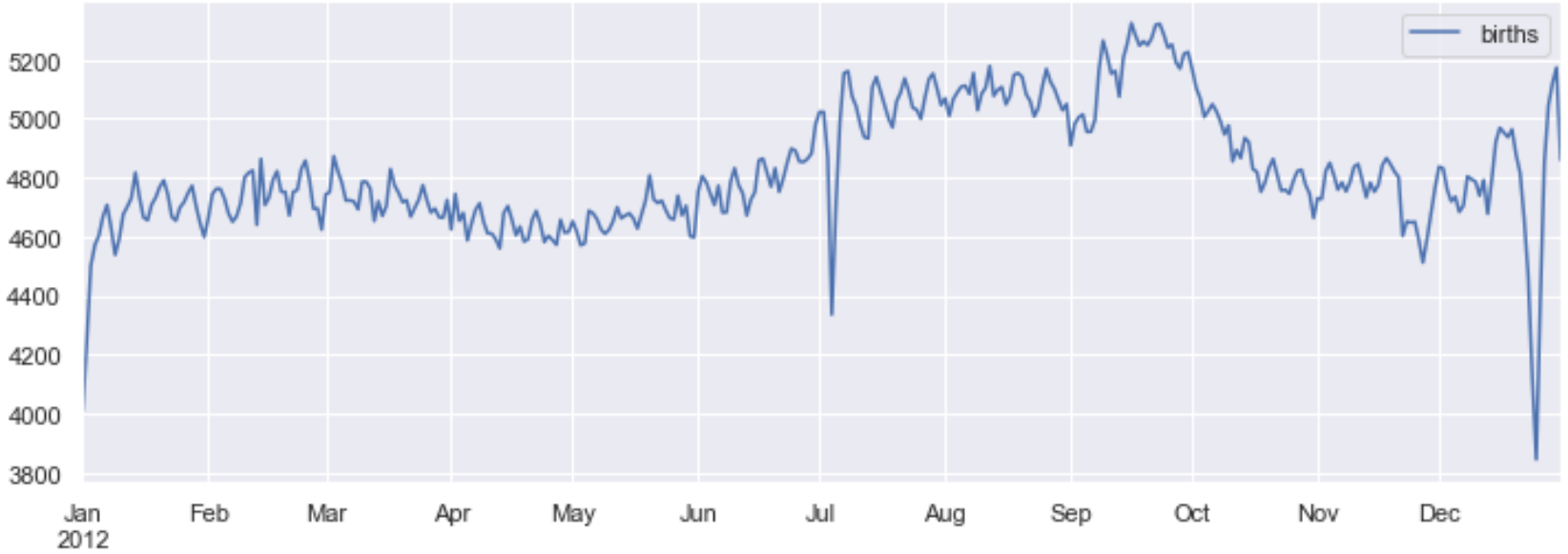
Out[16]:

| | births |
|---|---|
| 2012-01-01 | 4009.225 |
| 2012-01-02 | 4247.400 |
| 2012-01-03 | 4500.900 |
| 2012-01-04 | 4571.350 |
| 2012-01-05 | 4603.625 |

# Pivot.plot

```
# Plot the results
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax);
```

In [17]:
```
# Plot the results
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax);
```



# matplotlib

Luís Garmendia

# matplotlib

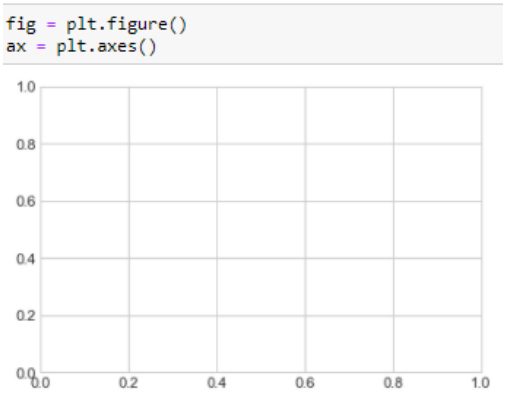
import the packages we will use

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```

# matplotlib

we start by creating a figure and an axes

```
fig = plt.figure()
ax = plt.axes()
```



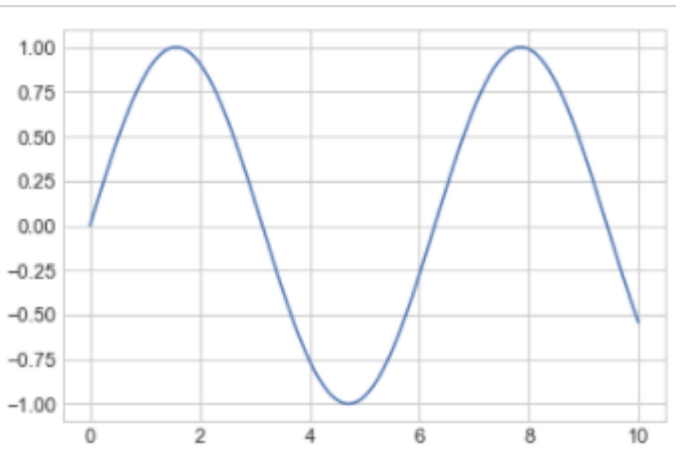

In Matplotlib, the *figure* (an instance of the class plt.Figure) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels.

The *axes* (an instance of the class plt.Axes) is what we see above: a bounding box with ticks and labels, which will eventually contain the plot elements that make up our visualization.

# matplotlib

Once we have created an axes, we can use the ax.plot function to plot some data. Let's start with a simple sinusoid

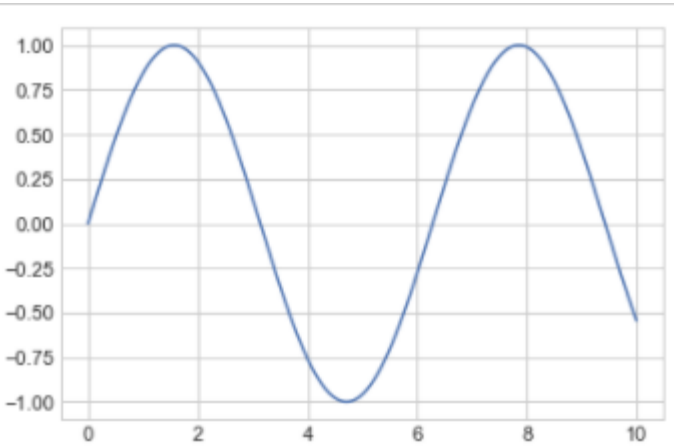
```
fig = plt.figure()
ax = plt.axes()

x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```



# Matplotlib
# plt.plot

Alternatively, we can use the pylab interface and let the figure and axes be created for us in the background

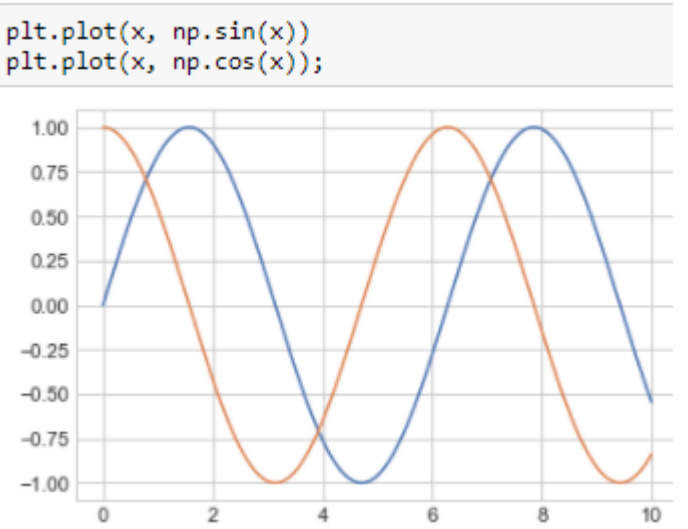
```
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x));
```



# matplotlib

If we want to create a single figure with multiple lines, we can simply call the plot function multiple times:

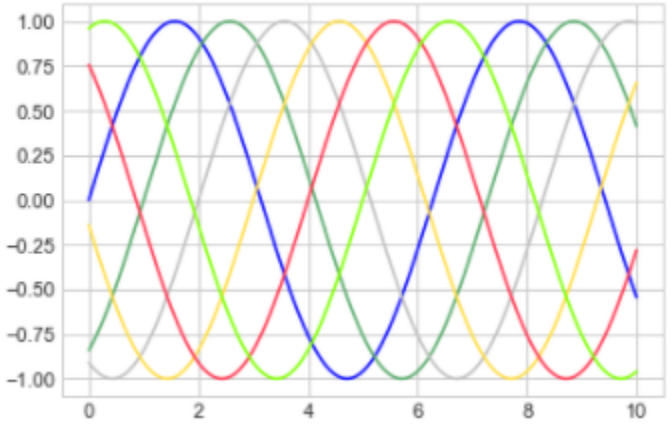
```
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```



# Matplotlib colors

If we want to create a single figure with multiple lines, we can simply call the plot function multiple times:

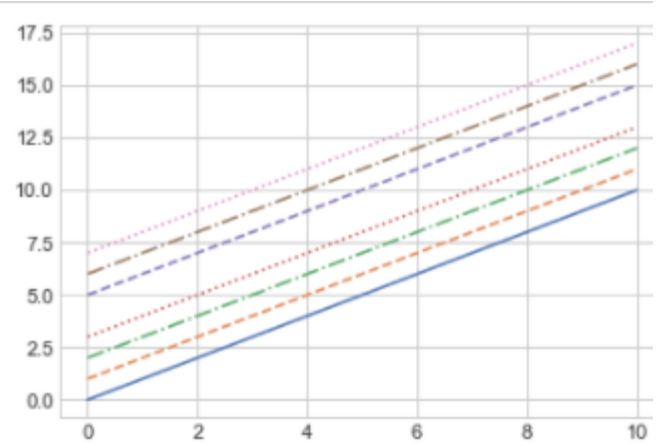
```
plt.plot(x, np.sin(x - 0), color='blue') # specify color by name
plt.plot(x, np.sin(x - 1), color='g') # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75') # Grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44') # Hex code (RRGGBB from 00 to FF)
plt.plot(x, np.sin(x - 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
```



# Matplotlib linestyle

Similarly, the line style can be adjusted using the linestyle keyword:

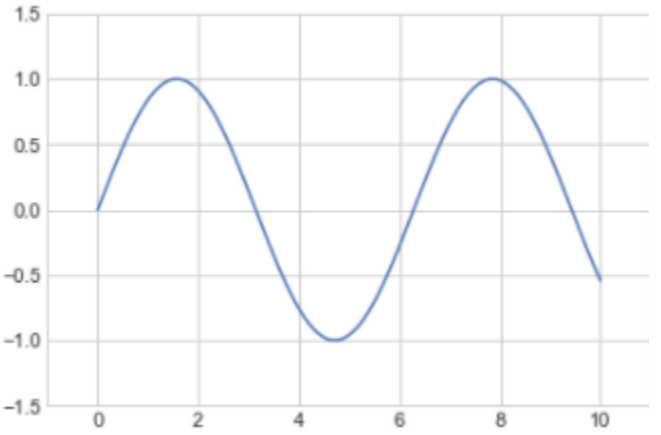
```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted'); # For short, you can use the following codes:

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

plt.plot(x, x + 0, '-g') # solid green
plt.plot(x, x + 1, '--c') # dashed cyan
plt.plot(x, x + 2, '-.k') # dashdot black
plt.plot(x, x + 3, ':r'); # dotted red
```



# Matplotlib
# Axex lims

The most basic way to adjust axis limits is to use the plt.xlim() and plt.ylim() methods:

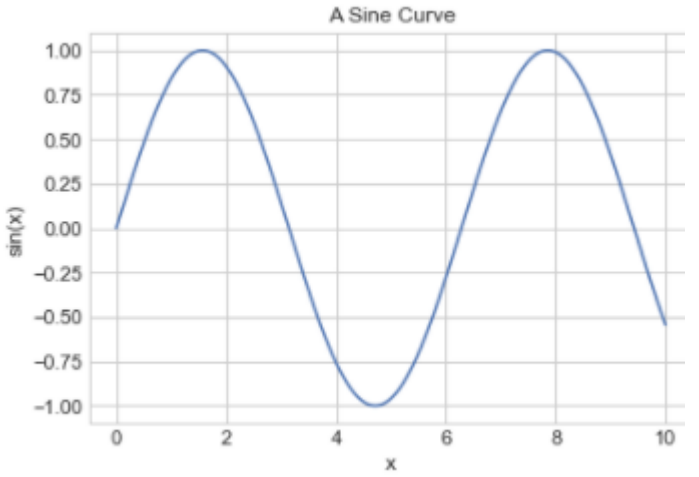
```
plt.plot(x, np.sin(x))
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```



# Matplotlib Labels

As the last piece of this section, we'll briefly look at the labeling of plots: titles, axis labels, and simple legends.

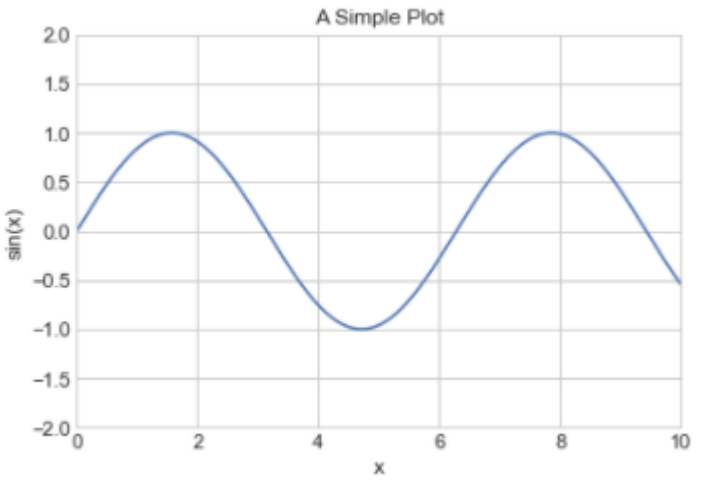
```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```



# Matplotlib
# .set

Use the ax.set() method to set all these properties at once:

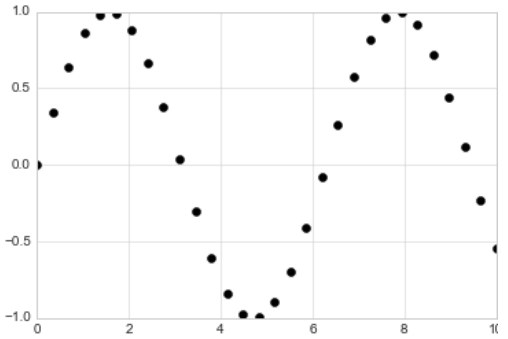
```
ax = plt.axes()
ax.plot(x, np.sin(x))
ax.set(xlim=(0, 10), ylim=(-2, 2), xlabel='x',
       ylabel='sin(x)', title='A Simple Plot');
```



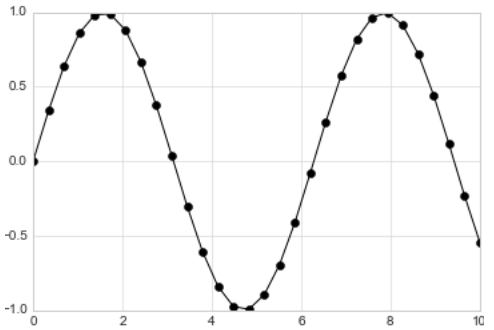
# Matplotlib scatter plot

It turns out that this same function can produce scatter plots as well:

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



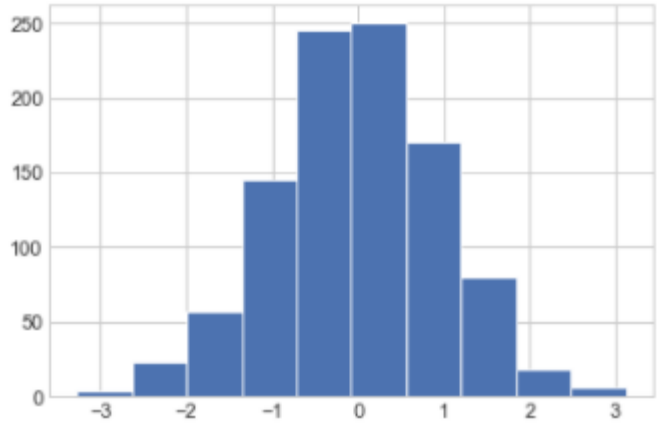

```
plt.plot(x, y, '-ok');
```



# Matplotlib histogram

Histogram:

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

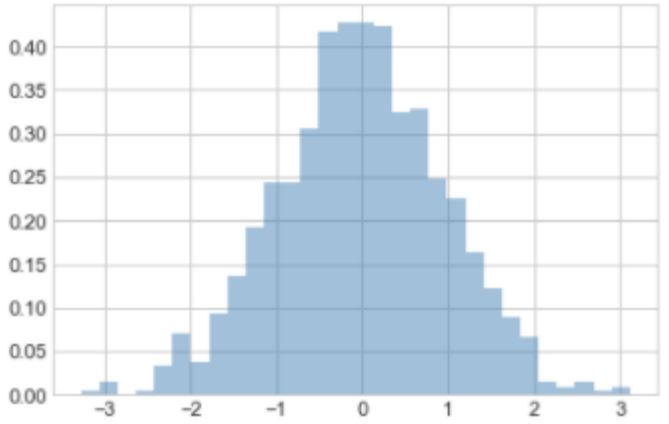


# Matplotlib histogram

Histogram:

The plt.hist docstring has more information on other customization options available. I find this combination of histtype='stepfilled' along with some transparency alpha to be very useful when comparing histograms of several distributions:

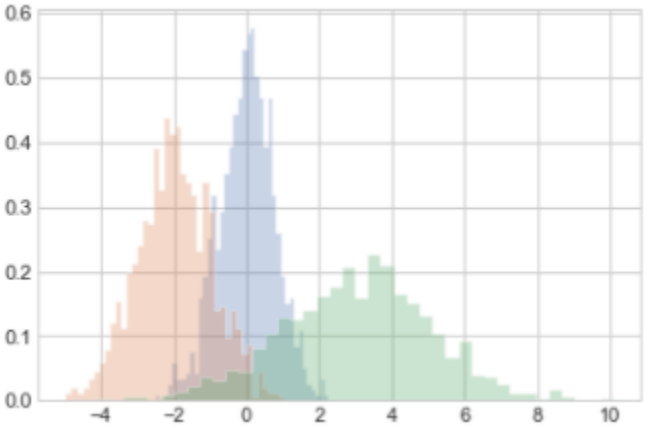
```
plt.hist(data, bins=30, normed=True, alpha=0.5, histtype='stepfilled',
color='steelblue', edgecolor='none');
```



# Matplotlib histogram

Histogram:

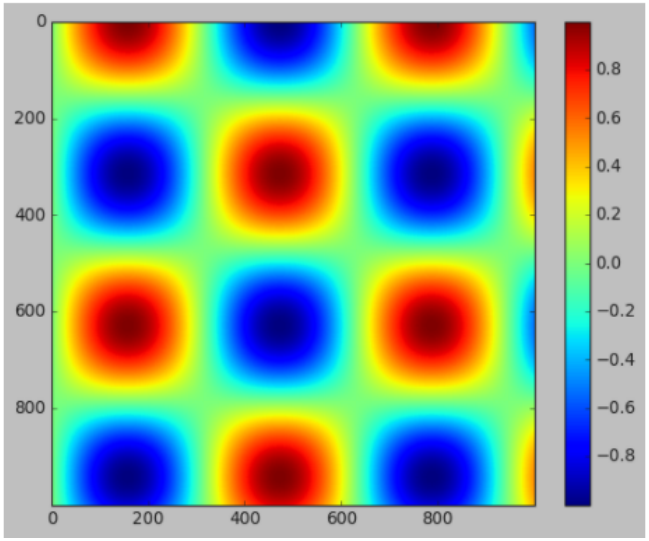
```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)
kwargs = dict(histtype='stepfilled', alpha=0.3, normed=True,
bins=40)
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```



# Matplotlib
# imshow and colorbar

Histogram:

```
plt.style.use('classic')
x = np.linspace(0, 10, 1000)
I = np.sin(x) * np.cos(x[:, np.newaxis])

plt.imshow(I)
plt.colorbar();
```

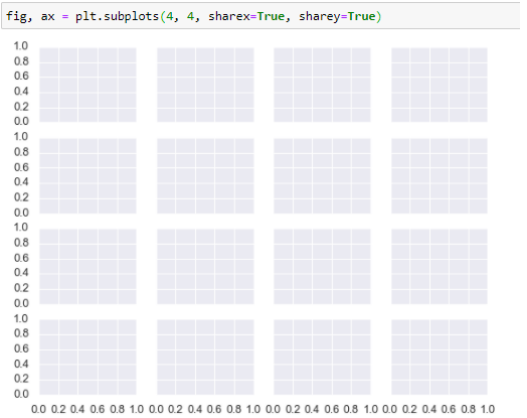



Explorar:
x.shape
x[:, np.newaxis]
I.shape

# Matplotlib subplots

One common problem with the default settings is that smaller subplots can end up with crowded labels. We can see this in the plot grid shown here:

```
fig, ax = plt.subplots(4, 4, sharex=True, sharey=True)
```




https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html

# Matplotlib tics

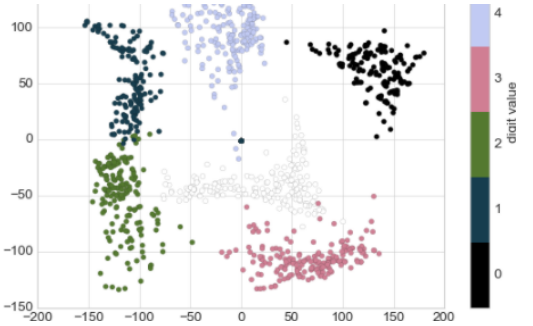
For now, let's start by downloading the digits data and visualizing several of the example images with plt.imshow():

```
# load images of the digits 0 through 5 and visualize several of them
plt.style.use('seaborn-whitegrid')
from sklearn.datasets import load_digits
digits = load_digits(n_class=6)
fig, ax = plt.subplots(8, 8, figsize=(6, 6))
for i, axi in enumerate(ax.flat):
    axi.imshow(digits.images[i], cmap='binary')
    axi.set(xticks=[], yticks=[])
```

# Matplotlib
# plt.scatter

For now, let's start by downloading the digits data and visualizing several of the example images with plt.imshow():

```
# project the digits into 2 dimensions using IsoMap
from sklearn.manifold import Isomap
iso = Isomap(n_components=2)
projection = iso.fit_transform(digits.data)

# plot the results
plt.scatter(projection[:, 0], projection[:, 1], lw=0.1,
            c=digits.target, cmap=plt.cm.get_cmap('cubehelix', 6))
plt.colorbar(ticks=range(6), label='digit value')
plt.clim(-0.5, 5.5)
```

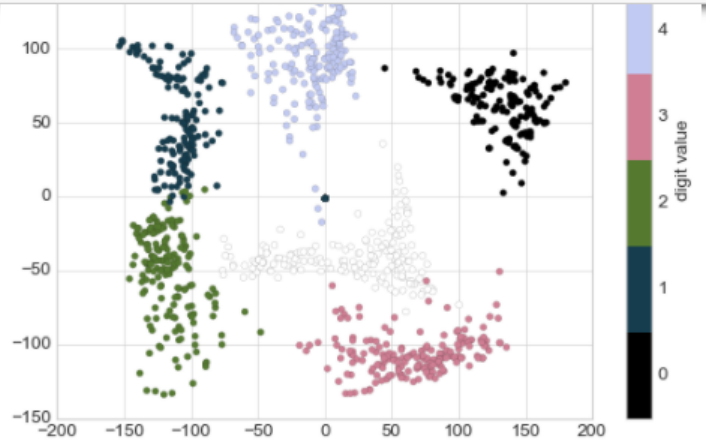



Explore projection

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html

# Matplotlib plt.scatter

```
In [543]: # project the digits into 2 dimensions using IsoMap
          from sklearn.manifold import Isomap
          iso = Isomap(n_components=2)
          projection = iso.fit_transform(digits.data)

In [545]: projection

Out[545]: array([[ 151.52041613,   55.39201656],
                 [ -97.77177886,   58.0822836 ],
                 [-112.70662015,  -45.22557972],
                 ...,
                 [   8.8746121 ,  111.16000848],
                 [  12.29531467,   95.65099338],
                 [ 106.8409937 ,   83.03555368]])

In [546]: # plot the results
          plt.scatter(projection[:, 0], projection[:, 1], lw=0.1,
                      c=digits.target, cmap=plt.cm.get_cmap('cubehelix', 6))
          plt.colorbar(ticks=range(6), label='digit value')
          plt.clim(-0.5, 5.5)
```




https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html

# Seaborn vs Matplotlib plt.style

Matplotlib team is addressing this: it has recently added the plt.style tools

```
import matplotlib.pyplot as plt
plt.style.use('classic')
%matplotlib inline
import numpy as np
import pandas as pd
```

# Seaborn vs Matplotlib plt.style

Now we create some random walk data:

```
# Create some data
rng = np.random.RandomState(0)
x = np.linspace(0, 10, 500)
y = np.cumsum(rng.randn(500, 6), 0)
```

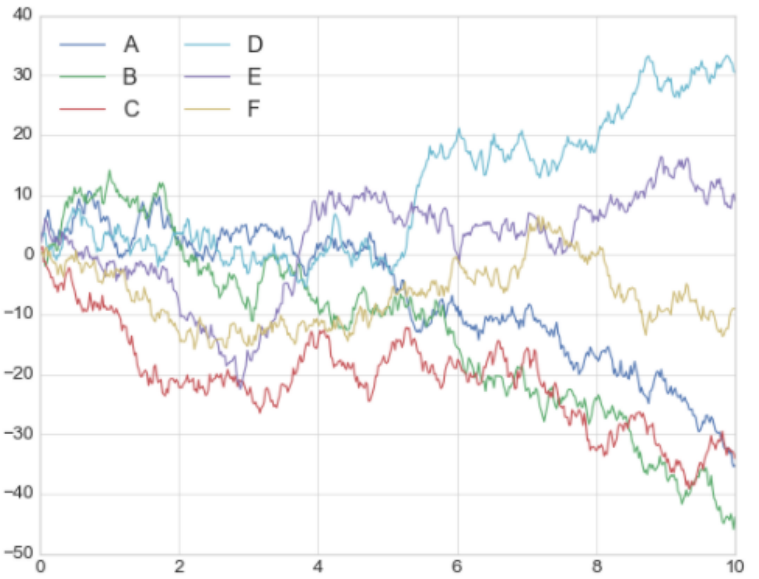
```
In [547]: # Create some data
          rng = np.random.RandomState(0)
          x = np.linspace(0, 10, 500)
          y = np.cumsum(rng.randn(500, 6), 0)

In [548]: x.shape

Out[548]: (500,)

In [549]: y.shape

Out[549]: (500, 6)
```

# Seaborn vs Matplotlib plt.style

*Plot the data with Matplotlib defaults*

```
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left');
```

```
In [550]: plt.plot(x, y)
          plt.legend('ABCDEF', ncol=2, loc='upper left');
```



# Seaborn vs Matplotlib plt.style

Now let's take a look at how it works with Seaborn. As we will see, Seaborn has many of its own high-level plotting routines, but it can also overwrite Matplotlib's default parameters and in turn get even simple Matplotlib scripts to produce vastly superior output. We can set the style by calling Seaborn's set() method. By convention, Seaborn is imported as sns:

```
import seaborn as sns
sns.set();
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left')
```

```
In [551]: import seaborn as sns
          sns.set();
          plt.plot(x, y)
          plt.legend('ABCDEF', ncol=2, loc='upper left')

Out[551]: <matplotlib.legend.Legend at 0x260aa666bc8>
```

# Seaborn vs Matplotlib plt.style

Now let's take a look at how it works with Seaborn. As we will see, Seaborn has many of its own high-level plotting routines, but it can also overwrite Matplotlib's default parameters and in turn get even simple Matplotlib scripts to produce vastly superior output. We can set the style by calling Seaborn's set() method. By convention, Seaborn is imported as sns:

```
import seaborn as sns
sns.set();
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left')
```

# Seaborn pandas pairplots

When you generalize joint plots to datasets of larger dimensions, you end up with *pair plots*. This is very useful for exploring correlations between multidimensional data,

```
iris = sns.load_dataset("iris")
iris.head()
```

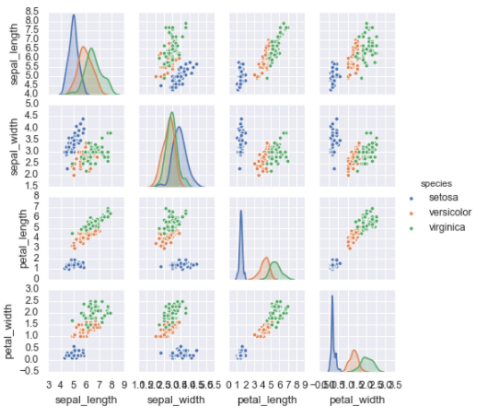
```
In [552]: iris = sns.load_dataset("iris")
          iris.head()
```

Out[552]:

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

```
sns.pairplot(iris, hue='species', size=2.5);
```

```
sns.pairplot(iris, hue='species', size=1.5);
```



# Other graphics libraries

Although Matplotlib is the most prominent Python visualization library, there are other more modern tools that are worth exploring as well. I'll mention a few of them briefly here:

•Bokeh is a JavaScript visualization library with a Python frontend that creates highly interactive visualizations capable of handling very large and/or streaming datasets. The Python front-end outputs a JSON data structure that can be interpreted by the Bokeh JS engine.

•Plotly is the eponymous open source product of the Plotly company, and is similar in spirit to Bokeh. Because Plotly is the main product of a startup, it is receiving a high level of development effort. Use of the library is entirely free.

•Vispy is an actively developed project focused on dynamic visualizations of very large datasets. Because it is built to target OpenGL and make use of efficient graphics processors in your computer, it is able to render some quite large and stunning visualizations.

•Vega and Vega-Lite are declarative graphics representations, and are the product of years of research into the fundamental language of data visualization. The reference rendering implementation is JavaScript, but the API is language agnostic.

•There is a Python API under development in the Altair package.

# References

https://ipython.org/

https://numpy.org/

https://jupyter.org/try

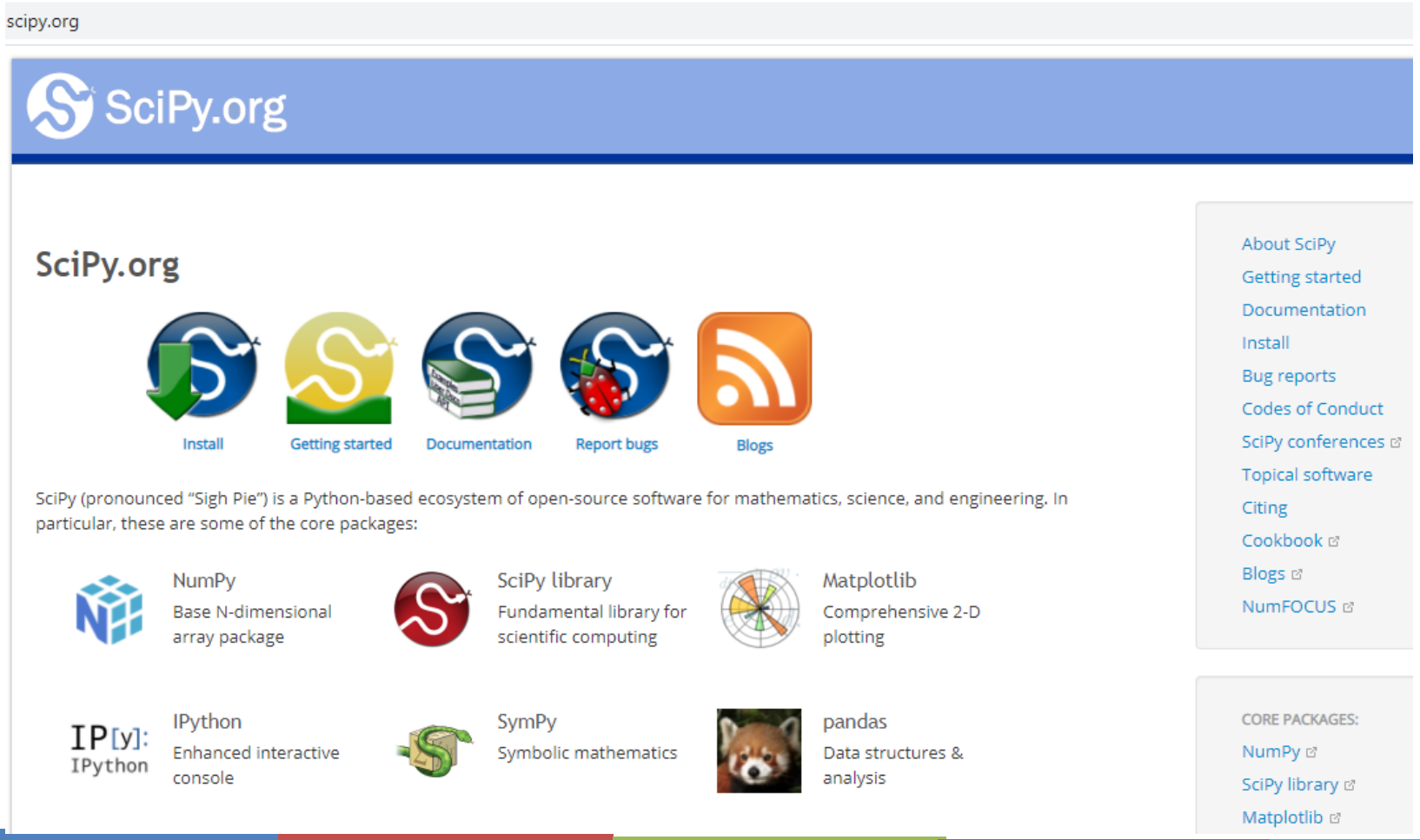
https://pandas.pydata.org/

http://seaborn.pydata.org/

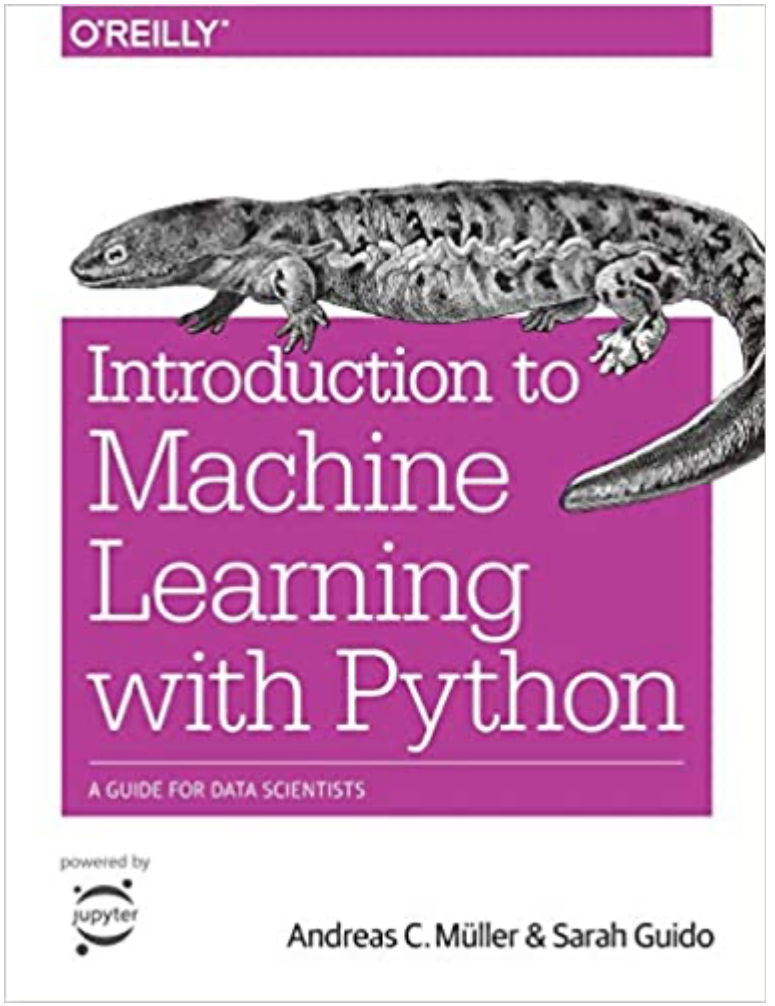
https://scikit-learn.org/stable/

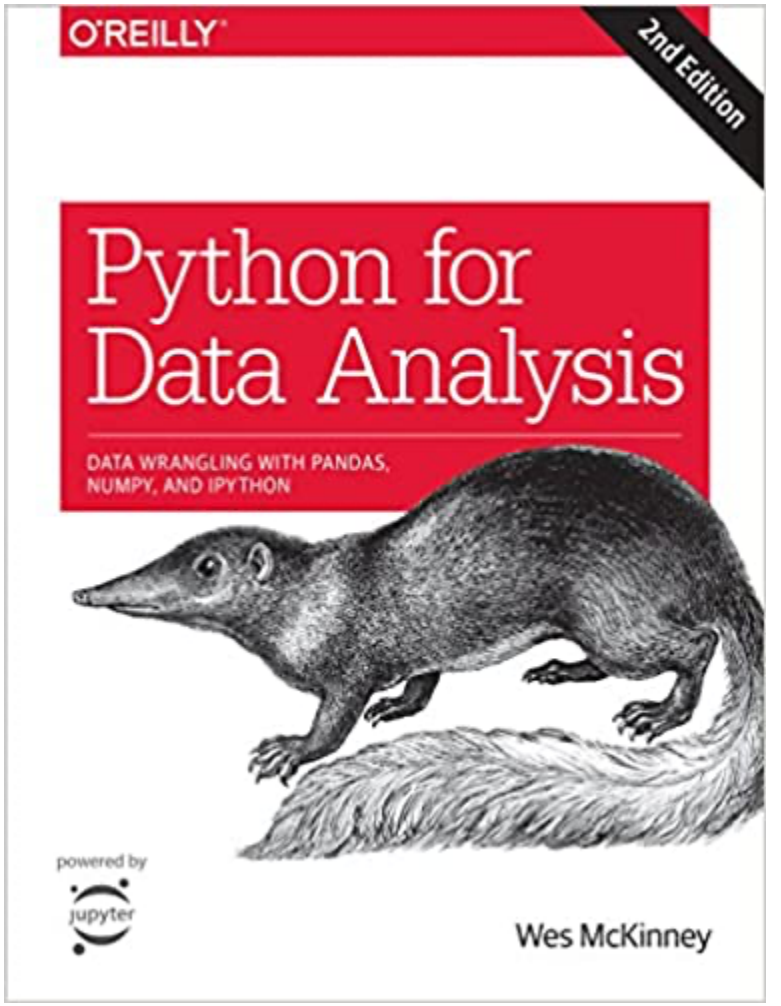
# References

https://www.scipy.org/

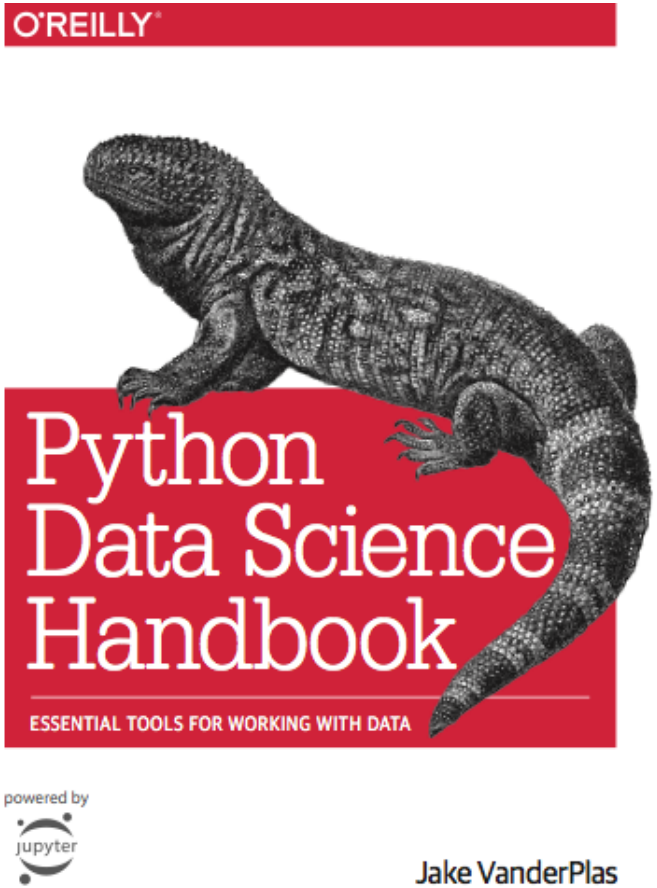


# References


O'REILLY
Introduction to
Machine
Learning
with Python
A GUIDE FOR DATA SCIENTISTS
powered by
jupyter
Andreas C. Müller & Sarah Guido


O'REILLY
2nd Edition
Python for
Data Analysis
DATA WRANGLING WITH PANDAS,
NUMPY, AND IPYTHON
powered by
jupyter
Wes McKinney

# References




https://jakevdp.github.io/PythonDataScienceHandbook/