

CURSO DE PANDAS Y NUMPY PARA DATA SCIENCE|

LIBRERÍAS DE MANIPULACIÓN DE DATOS CON PYTHON	2
¿POR QUÉ NUMPY Y PANDAS?	2
NUMPY	6
NUMPY ARRAY	6
TIPOS DE DATOS.....	10
DIMENSIONES	12
CREANDO ARRAYS	18
SHAPE Y RESHAPE	21
FUNCIONES PRINCIPALES DE NUMPY	24
COPY	28
CONDICIONES.....	30
OPERACIONES	31
QUIZ NUMPY.....	34
PANDAS.....	34
SERIES Y DATAFRAMES EN PANDAS	34
LEER ARCHIVOS CSV Y JSON CON PANDAS.....	38
FILTRADO CON LOC Y ILOC	43
AGREGAR O ELIMINAR DATOS CON PANDAS	46
MANEJO DE DATOS NULOS	47
FILTRADO POR CONDICIONES	50
FUNCIONES PRINCIPALES DE PANDAS	52
GROUPBY	55
COMBINANDO DATAFRAMES	57
MERGE Y CONCAT.....	61
JOIN	66
PIVOT Y MELT	67
APPLY	71
CIERRE	72
POSIBILIDADES CON PANDAS Y NUMPY.....	72

LIBRERÍAS DE MANIPULACIÓN DE DATOS CON PYTHON

¿POR QUÉ NUMPY Y PANDAS?

Vamos a aprender de 2 librerías muy importantes para la manipulación en la ciencia de datos (**Numpy y Pandas**)

¿Por qué NumPy?

Es una librería enfocada al cálculo numérico y manejo de Arrays.

- Es muy veloz, hasta 50 veces más rápido que usar una lista de Python o C.
- Optimiza el almacenamiento en memoria.
- Maneja distintos tipos de datos.

Es una librería muy poderosa, se pueden crear **redes neuronales** desde cero.



¿Por qué Pandas?

Pandas está enfocada a la manipulación y análisis de datos.

- Al estar construido sobre NumPy es veloz.
- Requiere poco código para manipular los datos.
- Soporta múltiples formatos de archivos.

- Ordena los datos en una alienación inteligente.

Se pueden manejar ***grandes cantidades de datos***, hacer analítica y generar dashboards.



La forma de importar estas librerías es de la siguiente manera:

```
import numpy as np
```

```
import pandas as pd
```

- **NumPy** = Cálculo matemático y manejo de arrays (listas o diccionarios). El nombre viene de "Numerical Python extensions".
- **Pandas** = Manipulación y análisis de datos. El nombre viene de "Panel data".

Numpy Cheat Sheet

PYTHON PACKAGE

CREATED BY: ARIANNE COLTON AND SEAN CHEN

NUMPY (NUMERICAL PYTHON)

What is NumPy?

Foundation package for scientific computing in Python

Why NumPy?

- NumPy **'ndarray'** is a much more efficient way of storing and manipulating **"numerical data"** than the built-in Python data structures.
- Libraries written in lower-level languages, such as C, can operate on data stored in NumPy **'ndarray'** without copying any data.

N-DIMENSIONAL ARRAY (NDARRAY)

What is NdArray?

Fast and space-efficient multidimensional array (container for homogeneous data) providing vectorized arithmetic operations

Create NdArray	<pre>np.array(seq1) # seq1 is any sequence like object, # i.e. [1, 2, 3]</pre>
Create Special NdArray	<pre>1, np.zeros(10) # one dimensional ndarray with 10 # elements of value 0 2, np.ones(2, 3) # two dimensional ndarray with 6 # elements of value 1 3, np.empty(3, 4, 5) * # three dimensional ndarray of # uninitialized values 4, np.eye(N) or np.identity(N) # creates N by N identity matrix</pre>
NdArray version of Python's range	<pre>np.arange(1, 10)</pre>
Get # of Dimension	<pre>ndarray.ndim</pre>
Get Dimension Size	<pre>dim1size, dim2size, ... = ndarray.shape</pre>
Get Data Type **	<pre>ndarray.dtype</pre>
Explicit Casting	<pre>ndarray2 = ndarray1. astype(np.int32) ***</pre>

- Cannot assume empty() will return all zeros. It could be garbage values.

- ** Default data type is **'np.float64'**. This is equivalent to Python's float type which is 8 bytes (64 bits); thus the name **'float64'**.
- *** If casting were to fail for some reason, **'TypeError'** will be raised.

SLICING (INDEXING/SUBSETTING)

- Slicing (i.e. `ndarray[2:6]`) is a **'view'** on the original array. **Data is NOT copied**. Any modifications (i.e. `ndarray[2:6] = 8`) to the **'view'** will be reflected in the original array.

- Instead of a **'view'**, explicit copy of slicing via:

```
ndarray[2:6].copy()
```

- Multidimensional array indexing notation:

```
ndarray[0][2] Or ndarray[0, 2]
```

* Boolean indexing :

```
ndarray[(names == 'Bob') | (names ==
'Will'), 2:]
# '2' means select from 3rd column on
```

- Selecting data by boolean indexing **ALWAYS** creates a copy of the data.
- The **'and'** and **'or'** keywords do NOT work with boolean arrays. Use **&** and **|**.

* Fancy indexing (aka 'indexing using integer arrays')

Select a subset of rows in a particular order :

```
ndarray[[ 3, 8, 4 ]
ndarray[[ -1, 6 ] ]
```

negative indices select rows from the end

- Fancy indexing **ALWAYS** creates a copy of the data.

NUMPY (NUMERICAL PYTHON)

Setting data with assignment :

```
ndarray[ndarray < 0] = 0 *
```

- If `ndarray` is two-dimensional, `ndarray < 0` creates a two-dimensional boolean array.

COMMON OPERATIONS

1. Transposing

- A special form of reshaping which returns a **'view'** on the underlying data without copying anything.

```
ndarray.transpose() or
ndarray.T or
ndarray.swapaxes(0, 1)
```

2. Vectorized wrappers (for functions that take scalar values)

- `math.sqrt()` works on only a scalar

```
np.sqrt(seq1) # any sequence (list, ndarray, etc) to return a ndarray
```

3. Vectorized expressions

- `np.where(cond, x, y)` is a vectorized version of the expression **'x if condition else y'**

```
np.where([True, False], (1, 2),
[2, 3]) => ndarray (1, 3)
```

* Common Usages :

```
np.where(matrixArray > 0, 1, -1)
=> a new array (same shape) of 1 or -1 values
np.where(cond, 1, 0).argmax() *
=> Find the first True element
```

- `argmax()` can be used to find the index of the maximum element. Example usage is find the first element that has a **'price > number'** in an array of price data.

4. Aggregations/Reductions Methods (i.e. mean, sum, std)

```
Compute mean ndarray.mean() or
np.mean(ndarray)
```

```
Compute statistics over axis* ndarray.mean(axis = 1)
ndarray.sum(axis = 0)
```

- axis = 0** means column axis, **1** is row axis.

5. Boolean arrays methods

Count # of 'Trues' in boolean array	<pre>(ndarray > 0).sum()</pre>
If at least one value is 'True'	<pre>ndarray.any()</pre>
If all values are 'True'	<pre>ndarray.all()</pre>

Note: These methods also work with non-boolean arrays, where non-zero elements evaluate to True.

6. Sorting

Inplace sorting	<pre>ndarray.sort()</pre>
Return a sorted copy instead of inplace	<pre>sorted1 = np.sort(ndarray)</pre>

7. Set methods

Return sorted unique values	<pre>np.unique(ndarray)</pre>
Test membership of ndarray1 values in [2, 3, 6]	<pre>resultBooleanArray = np.in1d(ndarray1, [2, 3, 6])</pre>

- Other set methods: `intersect1d()`, `union1d()`, `setdiff1d()`, `setxor1d()`

8. Random number generation (np.random)

- Supplements the built-in Python `random *` with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions.

```
samples = np.random.normal(size = (3, 3))
```

- Python built-in `random` **ONLY** samples one value at a time.

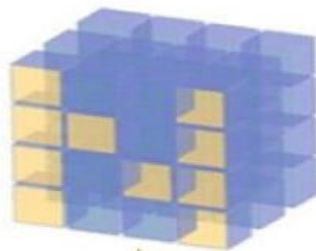
Created by Arienne Colton and Sean Chen

www.data-science-free.com

Based on content from

'Python for Data Analysis' by Wes McKinney

Updated: August 18, 2016



NumPy

NUMerical

PYthon

Potente estructuras de datos

Implementa matrices y matrices multidimensionales

Estas estructuras garantizan cálculos eficientes con matrices

Data Wrangling with pandas Cheat Sheet

<http://pandas.pydata.org>

Syntax – Creating DataFrames

	a	b	c
1	4	5	6
2	7	8	9
3	10	11	12

```
df = pd.DataFrame({
    "a": [4, 5, 6],
    "b": [7, 8, 9],
    "c": [10, 11, 12],
    index = [1, 2, 3]
})
```

Specify values for each column.

```
df = pd.DataFrame([
    [4, 7, 10],
    [5, 8, 11],
    [6, 9, 12]],
    index=[1, 2, 3],
    columns=['a', 'b', 'c'])
```

Specify values for each row.

	a	b	c
1	4	5	6
2	7	8	9
3	10	11	12

```
df = pd.DataFrame({
    "a": [4, 5, 6],
    "b": [7, 8, 9],
    "c": [10, 11, 12]},
    index = pd.MultiIndex.from_tuples(
        [(1, 'd'), (2, 'e'), (3, 'f')],
        names=['n', 'v']))
```

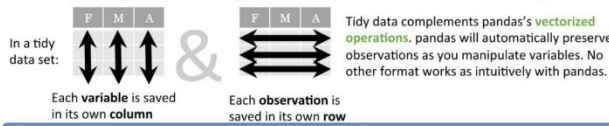
Create DataFrame with a MultiIndex

Method Chaining

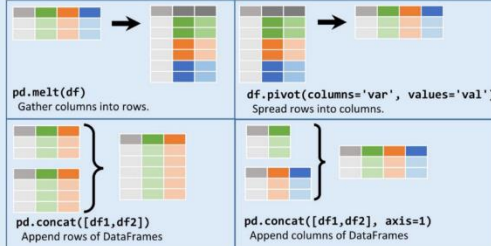
Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

```
df = df.melt(df)
    .rename(columns={
        'variable': 'var',
        'value': 'val'})
    .query('val >= 200')
```

Tidy Data – A foundation for wrangling in pandas



Reshaping Data – Change the layout of a data set



```
df.sort_values('mpg')
    Order rows by values of a column (low to high).

df.sort_values('mpg', ascending=False)
    Order rows by values of a column (high to low).

df.rename(columns = {'y': 'year'})
    Rename the columns of a DataFrame

df.sort_index()
    Sort the index of a DataFrame

df.reset_index()
    Reset index of DataFrame to row numbers, moving index to columns.

df.drop(['Length', 'Height'], axis=1)
    Drop columns from DataFrame
```

Subset Observations (Rows)

df[df.Length > 7]

Extract rows that meet logical criteria.

df.drop_duplicates()

Remove duplicate rows (only considers columns).

df.head(n)

Select first n rows.

df.tail(n)

Select last n rows.

df.sample(frac=0.5)

Randomly select fraction of rows.

df.sample(n=10)

Randomly select n rows.

df.iloc[10:20]

Select rows by position.

df.nlargest(n, 'value')

Select and order top n entries.

df.nsmallest(n, 'value')

Select and order bottom n entries.

Subset Variables (Columns)

df[['width', 'length', 'species']]

Select multiple columns with specific names.

df['width'] or df.width

Select single column with specific name.

df.filter(regex='regex')

Select columns whose name matches regular expression regex.

regex (Regular Expressions) Examples

regex	Matches
'.'	Matches strings containing a period '.'
'Length\$'	Matches strings ending with word 'Length'
'^Sepal'	Matches strings beginning with the word 'Sepal'
'^x[1-5]'	Matches strings beginning with 'x' and ending with 1,2,3,4,5
'^(?!Species)\$'	Matches strings except the string 'Species'

df.loc[:, 'x2': 'x4']

Select all columns between x2 and x4 (inclusive).

df.iloc[:, 1, 2, 5]

Select columns in positions 1, 2 and 5 (first column is 0).

df.loc[df['a'] > 10, ['a', 'c']]

Select rows meeting logical condition, and only the specific columns.

Summarize Data

```
df['w'].value_counts()
    Count number of rows with each unique value of variable

len(df)
    # of rows in DataFrame.

df['w'].nunique()
    # of distinct values in a column.

df.describe()
    Basic descriptive statistics for each column (or GroupBy)
```

pandas provides a large set of **summary functions** that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

sum()	Sum values of each object.	min()	Minimum value in each object.
count()	Count non-NA/null values of each object.	max()	Maximum value in each object.
median()	Median value of each object.	mean()	Mean value of each object.
quantile([0.25, 0.75])	Quantiles of each object.	var()	Variance of each object.
apply(function)	Apply function to each object.	std()	Standard deviation of each object.

Group Data

df.groupby(by="col")

Return a GroupBy object, grouped by values in column named "col".

df.groupby(level="ind")

Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group. Additional GroupBy functions:

size()

Size of each group.

agg(function)

Aggregate group using function.

Handling Missing Data

```
df.dropna()
    Drop rows with any column having NA/null data.

df.fillna(value)
    Replace all NA/null data with value.
```

Make New Columns

df.assign(Area=lambda df: df.Length*df.Height)

Compute and append one or more new columns.

df['Volume'] = df.Length*df.Height*df.Depth

Add single column.

pd.qcut(df.col, n, labels=False)

Bin column into n buckets.

pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

max(axis=1)	Element-wise max.	min(axis=1)	Element-wise min.
clip(lower=-10, upper=10)	Trim values at input thresholds	abs()	Absolute value.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

shift(1)	Copy with values shifted by 1.	shift(-1)	Copy with values lagged by 1.
rank(method='dense')	Ranks with no gaps.	cumsum()	Cumulative sum.
rank(method='min')	Ranks. Ties get min rank.	cummax()	Cumulative max.
rank(pct=True)	Ranks rescaled to interval [0, 1].	cumin()	Cumulative min.
rank(method='first')	Ranks. Ties go to first value.	cumprod()	Cumulative product.

Windows

```
df.expanding()
    Return an Expanding object allowing summary functions to be applied cumulatively.

df.rolling(n)
    Return a Rolling object allowing summary functions to be applied to windows of length n.
```

Plotting

df.plot.hist()

Histogram for each column

df.plot.scatter(x='w', y='h')

Scatter chart using pairs of points

Combine Data Sets

adf + bdf =

Standard Joins

pd.merge(adf, bdf, how='left', on='x1')

Join matching rows from bdf to adf.

pd.merge(adf, bdf, how='right', on='x1')

Join matching rows from adf to bdf.

pd.merge(adf, bdf, how='inner', on='x1')

Join data. Retain only rows in both sets.

pd.merge(adf, bdf, how='outer', on='x1')

Join data. Retain all values, all rows.

Filtering Joins

adf[adf.x1.isin(bdf.x1)]

All rows in adf that have a match in bdf.

adf[~adf.x1.isin(bdf.x1)]

All rows in adf that do not have a match in bdf.

ydf + zdf =

Set-like Operations

pd.merge(ydf, zdf)

Rows that appear in both ydf and zdf (Intersection).

pd.merge(ydf, zdf, how='outer')

Rows that appear in either or both ydf and zdf (Union).

pd.merge(ydf, zdf, how='outer', indicator=True)

.query('.merge == "left_only")

.drop(['_merge'], axis=1)

Rows that appear in ydf but not zdf (Setdiff).

NUMPY

NUMPY ARRAY

El array es el principal objeto de la librería. Representa datos de manera estructurada y se puede acceder a ellos a través del indexado, a un dato específico o un grupo de muchos datos específicos.

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
lista
```

```
---> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Volvemos nuestra lista, un array

```
arr = np.array(lista)
```

```
arr
```

```
---> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Una matriz son varios **Vectores** o **listas** agrupadas una encima de la otra, es como una tabla de Excel

```
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
matriz = np.array(matriz)
```

```
matriz
```

```
---> array([[1, 2, 3],
```

```
         [4, 5, 6],
```

```
         [7, 8, 9]])
```

El **indexado** nos permite acceder a los elementos de los array y matrices

Los elementos se **empiezan a contar desde 0**.

```
arr[0]
```

```
---> 1
```

Index	0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8	9

Es posible **operar** directamente con los elementos.

`arr[0] + arr[5]`

---> 7

0	+	5
1		6

En el caso de las **matrices**, al indexar una posición se regresa el array de dicha posición.

`matriz[0]`

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

Index	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Para seleccionar un solo elemento de la matriz se especifica la posición del elemento **separada por comas**.

Nota: El primer elemento selecciona las filas, el segundo elemento las columnas

`matriz[0, 2]`

```
---> 3
```

Slicing

Nos permite extraer varios datos, tiene un comienzo y un final.
En este ejemplo se está extrayendo datos desde la posición 1 hasta la 5.

```
arr[1:6]
```

```
---> array([2, 3, 4, 5, 6])
```

Si no se ingresa el **valor de inicio**, se toma el inicio como la posición 0.

```
arr[:6]
```

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

En cambio, si no se le da una **posición final**, se regresan todos los elementos hasta el final del array.

```
arr[2:]
```

```
---> array([3, 4, 5, 6, 7, 8, 9])
```

También se puede **trabajar por pasos**.

En este ejemplo de 3 en 3.

Regresa la posición 0, 0 + 3, 3 + 3 y como no hay posición 6 + 3, no se regrese nada.

```
arr[::3]
```

```
---> array([1, 4, 7])
```

Index	0	3	7
0	1	4	7

Cuando se le asigna un **valor negativo** se regresan los valores comenzando desde la última posición del array.

```
arr[-1]
```

```
---> 9
```

```
arr[-3:]
```

```
---> array([7, 8, 9])
```


1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Para el caso de las matrices, sucede algo similar.
Para acceder a los valores entre filas.

```
matriz[1:]
---> array([[4, 5, 6],
           [7, 8, 9]])
```

Para acceder a los valores entre filas y columnas.

```
matriz[1:, 0:2]
---> array([[4, 5],
           [7, 8]])
```

Reto

Crea una matriz de 3 dimensiones y cuéntanos:

- ¿Qué pudiste hacer?
- ¿Cómo hacer un Slicing de los datos?

Indexación

La indexación de matrices se utiliza para acceder a elementos especificando sus índices dentro del array.

arrays: En este tipo de dato todos los elementos tienen que ser del mismo tipo. El tipo se indica mediante el atributo *typecode*. Están definidas en el módulo nativo de *array*. El índice de los elementos es numérico y empieza en cero.

ndarray: Son tipos de datos tipo vector definidas en la librería *numpy*. En estas secuencias todos los elementos deben ser del mismo tipo. El tipo de los objetos se define con el atributo *dtype*. Si este parámetro no se especifica, por defecto considera el tipo flotante. El índice de los elementos es numérico y empieza en cero.

|

Fuente: Algoritmos Genéticos con Python.

Slicing

La operación de *slicing* consiste en acceder a la vez a varios elementos de una secuencia mediante los índices. De forma genérica, en una secuencia en Python la sintaxis para realizar la operación es *secuencia[i:j]*, accediendo a los elementos (i,j). La diferencia principal entre las secuencias es el objeto que devuelve la operación de *Slicing*. En particular, la gran diferencia reside en los arrays de numpy, ya que estos devuelven una vista de los elementos del array original.

Podemos hacer slice entre índices de la siguiente forma: [start:end]

Y también podemos definir los pasos, así: [start:end:step]

|

Fuente: Algoritmos Genéticos con Python.

TIPOS DE DATOS

Los arrays de NumPy solo pueden contener un tipo de dato, ya que esto es lo que le confiere las ventajas de la **optimización de memoria**.

Podemos conocer el tipo de datos del array consultando la propiedad `.dtype`

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

```
arr.dtype
```

```
---> dtype('int64')
```

Si queremos usar otro tipo de dato, lo podemos definir en la declaración del array.

```
arr = np.array([1, 2, 3, 4], dtype = 'float64')
```

```
arr.dtype
```

```
---> dtype('float64')
```

Ahora vemos que los valores están con punto decimal.

```
arr
```

```
---> array([1., 2., 3., 4.]
```

Si ya se tiene el array definido, se utiliza el método `.astype()` para convertir el tipo de dato.

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

```
arr = arr.astype(np.float64)
```

```
arr
```

```
---> array([1., 2., 3., 4.])
```

También se puede cambiar a **tipo booleano** recordando que los números diferentes de 0 se convierten en True.

```
arr = np.array([0, 1, 2, 3, 4])
```

```
arr = arr.astype(np.bool_)
```

```
arr
```

```
---> array([False,  True,  True,  True,  True])
```

También podemos convertir los datos en tipo **string**.

```
arr = np.array([0, 1, 2, 3, 4])
```

```
arr = arr.astype(np.string_)
```

```
arr
```

```
---> array([b'0', b'1', b'2', b'3', b'4'], dtype='<S21')
```

De igual manera, se puede pasar de string a número.

```
arr = np.array(['0', '1', '2', '3', '4'])
```

```
arr = arr.astype(np.int8)
```

```
arr
```

```
---> array([0, 1, 2, 3, 4], dtype=int8)
```

Si un elemento **no es de tipo número, el método falla**.

```
arr = np.array(['hola', '0', '1', '2', '3', '4'])
```

```
arr = arr.astype(np.int8)
```

```
arr
```

```
---> ValueError: invalid literal for int() with base 10: 'hola'
```

El array de Numpy únicamente puede tener un único tipo de datos en el cual va a trabajar. No puedo tener la mitad del array en **int** y la otra mitad en **bool**.

DIMENSIONES

Con las matrices podemos crear varias dimensiones, vamos a nombrarlas

- **Scalar:** 0 Un solo dato o valor

0

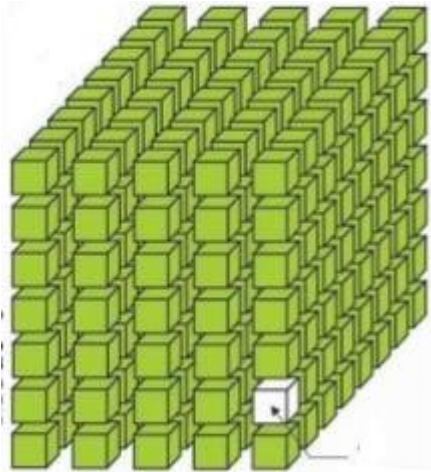
- **Vector:** Listas de Python

0	1	2	3	4
---	---	---	---	---

- **Matriz:** Hoja de cálculo

Color	País	Edad	Fruta
Rojo	España	24	Pera
Amarillo	Colombia	30	Manzana

- **Tensor:** Series de tiempo o Imágenes



Declarando un escalar.

.ndim Nos muestra las dimensiones que tiene

```
scalar = np.array(42)
```

```
print(scalar)
```

```
scalar.ndim
```

```
---> 42
```

```
---> 0
```

Declarando un vector.

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

```
print(vector)
```

```
vector.ndim
```

```
---> [1 2 3]
```

```
---> 1
```

Declarando una matriz.

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

```
print(matriz)
```

```
matriz.ndim
```

```
----[[1 2 3]
```

```
    [4 5 6]]
```

```
---> 2
```

Declarando un tensor.

```
tensor = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]], [[13, 13, 15], [16, 17, 18], [19, 20, 21], [22, 23, 24]]])
```

```
print(tensor)
```

```
tensor.ndim
```

```
---> [[[ 1  2  3]
```

```
      [ 4  5  6]
```

```
      [ 7  8  9]
```

```
      [10 11 12]]
```

```
      [[13 13 15]
```

```
      [16 17 18]
```

```
      [19 20 21]
```

```
      [22 23 24]]]
```

```
---> 3
```

Agregar o eliminar dimensiones

Se puede definir el **número de dimensiones** desde la declaración del array

```
vector = np.array([1, 2, 3], ndmin = 10)
```

```
print(vector)
```

```
vector.ndim
```

```
---> [[[[[[[[[[1 2 3]]]]]]]]]]]
```

```
---> 10
```

Se pueden expandir dimensiones a los array ya existentes con `expand_dims()`. **Axis = 0** hace referencia a las filas, mientras que **axis = 1** a las columnas.

```
expand = np.expand_dims(np.array([1, 2, 3]), axis = 0)
print(expand)
expand.ndim
```

```
---> [[1 2 3]]
```

```
---> 2
```

Remover/comprimir las dimensiones que no están siendo usadas.

```
print(vector, vector.ndim)
vector_2 = np.squeeze(vector)
print(vector_2, vector_2.ndim)
```

```
---> [[[[[[[[[1 2 3]]]]]]]]]] 10
```

```
---> [1 2 3] 1
```

Reto

1. Definir un tensor de 5D
2. Sumarle una dimensión en cualquier eje
3. Borrar las dimensiones que no se usen

Cuéntanos, ¿Cómo te fue y cómo lo solucionaste?

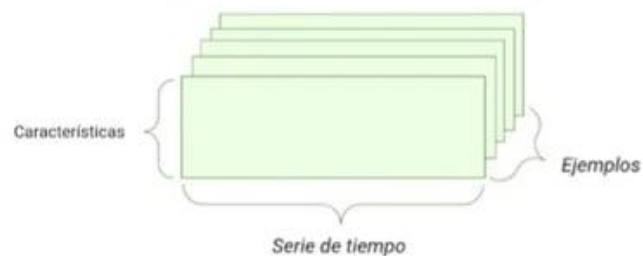
Matrix - 2D

(Ejemplos, características)

	A	B	C	D	E	F
1	Order ID	Product	Category	Amount	Date	Country
2	1	Carrots	Vegetables	\$4,270	1/6/2012	United States
3	2	Broccoli	Vegetables	\$8,239	1/7/2012	United Kingdom
4	3	Banana	Fruit	\$617	1/8/2012	United States
5	4	Banana	Fruit	\$8,384	1/10/2012	Canada
6	5	Beans	Vegetables	\$3,626	1/10/2012	Germany
7	6	Orange	Fruit	\$3,610	1/11/2012	United States
8	7	Broccoli	Vegetables	\$9,062	1/11/2012	Australia
9	8	Banana	Fruit	\$6,906	1/16/2012	New Zealand
10	9	Apple	Fruit	\$2,417	1/16/2012	France
11	10	Apple	Fruit	\$7,431	1/16/2012	Canada
12	11	Banana	Fruit	\$8,250	1/16/2012	Germany
13	12	Broccoli	Vegetables	\$7,012	1/18/2012	United States
14	13	Carrots	Vegetables	\$1,903	1/20/2012	Germany

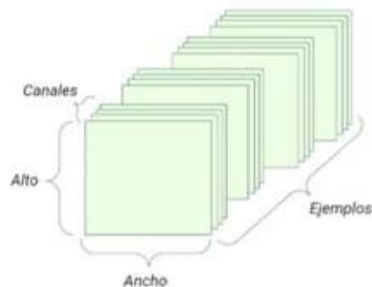
Tensor - 3D

(Ejemplos, Serie de tiempo, Características)



Tensor - 4D - Images

(Ejemplos, Ancho, Alto, Canales de color)



1. Escalar:

Un escalar es un valor único que representa una cantidad. Ejemplos de escalares incluyen temperatura, masa y distancia. En análisis de datos, los escalares se utilizan para representar puntos de datos individuales que solo tienen un valor asociado con ellos. Por ejemplo, la edad de una persona se puede representar como un valor escalar.

2. Vector:

Un vector es una cantidad que tiene magnitud y dirección. Ejemplos de vectores incluyen velocidad, fuerza y desplazamiento. En análisis de datos, los vectores se utilizan para representar puntos de datos que tienen múltiples valores asociados con ellos. Por ejemplo, las características de un conjunto de datos se pueden representar como un vector.

3. Matriz:

Una matriz es una matriz rectangular de números que se puede utilizar para representar datos. En análisis de datos, las matrices a menudo se utilizan para representar conjuntos de datos que tienen múltiples características y observaciones. Por ejemplo, un conjunto de datos que contiene la altura y el peso de un grupo de personas se puede representar como una matriz.

4. Tensor 3D:

Un tensor 3D es una cantidad que tiene tres índices y se puede utilizar para representar datos con tres dimensiones. Los tensores 3D se utilizan para representar datos en muchos campos, como física, ingeniería y gráficos por computadora. Un ejemplo de un tensor 3D podría ser una imagen 3D con canales de color RGB.

5. Tensor 4D:

Un tensor 4D es una cantidad que tiene cuatro índices y se puede utilizar para representar datos con cuatro dimensiones. Un ejemplo de un tensor 4D podría ser un video con canales de color RGB y una dimensión de tiempo.

6. Otros tensores:

Existen muchos otros tipos de tensores, incluidos tensores de orden superior, que tienen más de cuatro índices, y tensores mixtos, que tienen una combinación de índices covariantes y contravariantes. Estos tensores se utilizan en campos avanzados de matemáticas y física, como la relatividad general y la mecánica cuántica.

CREANDO ARRAYS

Numpy nos da varios métodos muy eficientes para poder crear arrays desde 0.

Este método de NumPy nos permite generar arrays sin definir previamente una lista.

```
np.arange(0,10)
```

```
---> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Un tercer argumento permite definir un tamaño de paso.

```
np.arange(0,20,2)
```

```
---> array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

np.zeros() Nos permite definir estructuras o esquemas.

```
np.zeros(3)
```

```
---> array([0., 0., 0.])
```

```
np.zeros((10,5))
```

```
---> array([[0., 0., 0., 0., 0.],  
          [0., 0., 0., 0., 0.],  
          [0., 0., 0., 0., 0.],  
          [0., 0., 0., 0., 0.],  
          [0., 0., 0., 0., 0.],  
          [0., 0., 0., 0., 0.],  
          [0., 0., 0., 0., 0.],  
          [0., 0., 0., 0., 0.],  
          [0., 0., 0., 0., 0.]])
```

De igual manera, tenemos np.ones()

```
np.ones(3)
```

```
---> array([1., 1., 1.])
```

np.linspace() Permite generar una array definiendo un inicio, un final y cuantas divisiones tendrá.

```
np.linspace(0, 10 , 10)
```

```
---> array([ 0.,1.11111111,2.22222222, 3.33333333, 4.44444444,  
          5.55555556, 6.66666667, 7.77777778, 8.88888889, 10.])
```

También podemos crear una matriz con una diagonal de 1 y el resto de 0.

```
np.eye(4)
```

```
----> array([[1., 0., 0., 0.],  
            [0., 1., 0., 0.],  
            [0., 0., 1., 0.],  
            [0., 0., 0., 1.]])
```

Otro método importante es generar números aleatorios.

```
np.random.rand()
```

```
---> 0.37185218178880153
```

También se pueden generar vectores.

```
np.random.rand(4)
```

```
---> array([0.77923054, 0.90495575, 0.12949965, 0.55974303])
```

Y a su vez generar matrices.

```
np.random.rand(4,4)
```

```
---> array([[0.26920153, 0.24873544, 0.02278515, 0.08250538],  
          [0.16755087, 0.59570639, 0.83604996, 0.57717126],  
          [0.00161574, 0.27857138, 0.33982786, 0.19693596],  
          [0.69474123, 0.01208492, 0.38613157, 0.609117  ]])
```

NumPy nos permite también generar números enteros.

En este caso números enteros entre el 1 y 14

```
np.random.randint(1,15)
```


```
---> 7
```


También podemos llevarlos a una estructura definida.


```
np.random.randint(1,15, (3,3))
```


```
---> array([[ 4,  2,  9],  
            [ 5,  7,  8],  
            [14, 14,  4]])
```

Creando Arrays

1  **np.arange** (Start,End,Steps) → es como el list(range(0,10)) pero como array


2  **np.zeros**(n)


3  **np.ones**(n)

4  **np.linspace**(Start, End, Cant n de Start a End)

5  **np.eye**(n) .. Matriz identidad

Arrays con numeros randoms

 **np.random.rand**(Columnas, Filas, mas dimensiones) .. Ambos con numeros randoms

 **np.random.randint**(Start, End, Dimensiones) .. N random entre Start y End y tupla dims

SHAPE Y RESHAPE

Hay 2 funciones muy importantes de los arreglos (Shape y Reshape). La forma de un arreglo nos va a decir con que **estructura** se está trabajando (tamaño, manipular, ingresar).

Shape

Indica la forma del arreglo.

```
arr = np.random.randint(1,10,(3,2))
```

```
arr.shape
```

```
---> (3, 2)
```

```
arr
```

```
---> array([[4, 2],  
           [4, 8],  
           [4, 3]])
```

Reshape

transforma el arreglo mientras se mantengan los elementos.

```
arr.reshape(1,6)
```

```
----> array([[4, 2, 4, 8, 4, 3]])
```

```
arr.reshape(2,3)
```

```
---> array([[4, 2, 4],  
           [8, 4, 3]])
```

```
np.reshape(arr,(1,6))
```

```
---> array([[4, 2, 4, 8, 4, 3]])
```

Se puede hacer un reshape como lo haría **C**.

```
np.reshape(arr,(2,3), 'C')
```

```
---> array([[4, 2, 4],  
           [8, 4, 3]])
```

También se puede hacer reshape a como lo haría **Fortran**.

```
np.reshape(arr,(2,3), 'F')
```

```
---> array([[4, 4, 8],  
           [4, 2, 3]])
```

Además, existe la opción de hacer reshape según como esté **optimizado nuestro computador**. En este caso es como en C.

```
np.reshape(arr,(2,3), 'A')
```

```
---> array([[4, 2, 4],  
           [8, 4, 3]])
```

No puedes cambiar la "forma" a la "forma" original del array, si tienes un (3,3) no lo puedes pasar a (4,2).

No respeta los 9 elementos del array original

Reto

- Crear un array de cualquier dimensión y cambiar sus dimensiones.
- Intenta cambiar el array de forma que no respete la estructura original

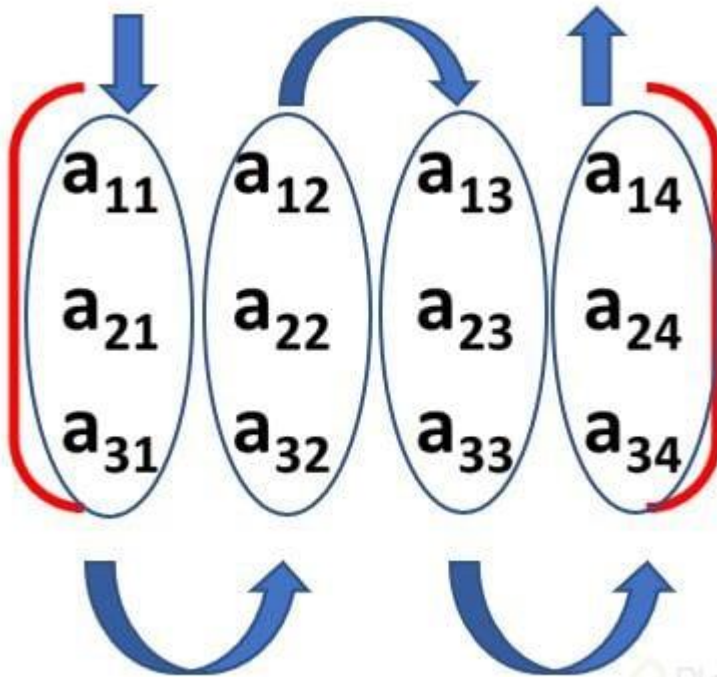
Forma*Como es la estructura del array*.... y reforma del array

```
✅ arr_shape = np.random.randint(Start, End,(3,2)) #Randoms entre 1  
y 10 en una matriz (n, n)
```

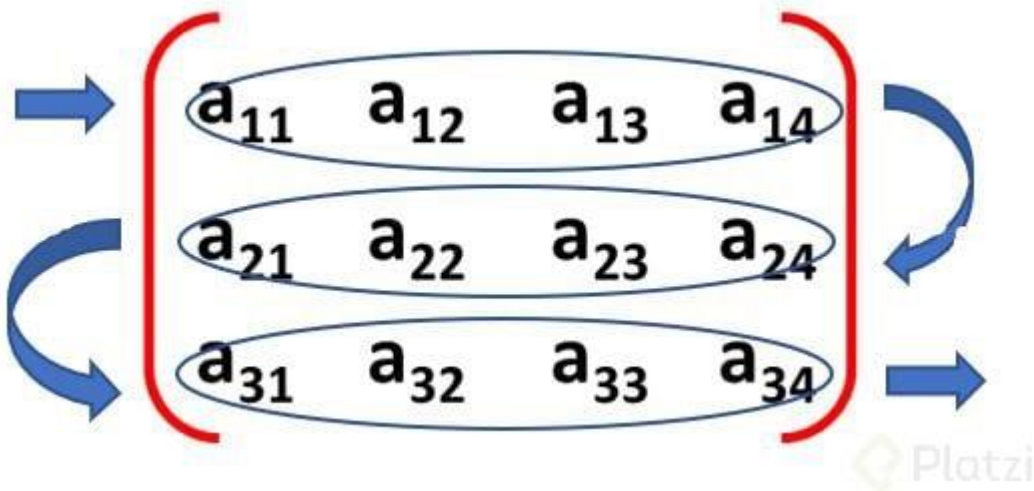
```
✅ arr_shape.reshape(dim, dim)
```


Una forma de ver lo que hace el Reshape según el argumento "C" o "F" es:

- Cuando apilamos los valores a través de Fortran, apilamos mediante columnas:



- Cuando apilamos los valores mediante el lenguaje C, apilamos mediante filas:



FUNCIONES PRINCIPALES DE NUMPY

Vamos a ver cuáles son las funciones que se utilizan normalmente con NumPy cuando analizamos los datos.

```
arr = np.random.randint(1, 20, 10)
```

```
arr
```

```
---> array([ 6, 11, 15, 12,  9, 17,  7,  7, 12,  3])
```

```
matriz = arr.reshape(2,5)
```

```
matriz
```

```
---> array([[ 6, 11, 15, 12,  9],  
          [17,  7,  7, 12,  3]])
```

.max Para el máximo

```
arr.max() ----> 17
```

```
matriz.max() ----> 17
```

Podemos regresar los máximos de cada fila o columna especificando el eje

Recuerda que:

0	Columnas
1	Filas

```
matriz.max(1) ---> array([15, 17])
```

```
matriz.max(0) ---> array([17, 11, 15, 12,  9])
```

También tenemos .argmax() que nos devuelve la posición del elemento

```
arr.argmax() ---> 9
```

En el caso de la matriz nos muestra con un 1 dónde se **encuentra el mayor** entre las columnas

```
matriz.argmax(0) ---> array([0, 1, 1, 0, 1])
```

De forma análoga tenemos .min()

```
arr.min() ---> 3
```

```
arr.argmin() ---> 3
```

```
matriz.min(0) ---> array([ 6,  7,  7, 12,  3])
```

```
matriz.argmin(1) ---> array([6, 3])
```

Podemos saber **la distancia entre** el valor más bajo con el más alto.

```
arr.ptp() # 17 - 3 ---> 14
```

```
matriz.ptp(0) ---> array([11,  4,  8,  0,  6])
```

Análisis estadístico

Ordenar los elementos:

```
arr.sort() ---> array([ 3,  6,  7,  7,  9, 11, 12, 12, 15, 17])
```

Obtener un percentil:

```
np.percentile(arr, 50) ---> 10.0
```

Mediana:

```
np.median(arr) ---> 10.0
```

Desviación estándar:

```
np.std(arr) ---> 4.0853396431631
```

Varianza:

```
np.var(arr) ---> 16.69
```

Promedio:

```
np.mean(arr) ---> 9.9
```

Lo mismo aplica para las matrices.

```
np.median(matriz, 1) ---> array([ 7., 12.])
```

Concatenación

Se pueden unir dos arrays

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

```
b= np.array([5, 6])  
np.concatenate((a,b), axis = 0)
```

---> ValueError: all the input arrays must have same number of dimensions, but the array at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(s)

El error anterior es debido a que 'a' tiene 2 dimensiones, mientras que 'b' tiene 1.

```
a.ndim ---> 2
```

```
b.ndim ---> 1
```

Debemos poner 'b' en 2 dimensiones también.

```
b = np.expand_dims(b, axis = 0)  
np.concatenate((a,b), axis = 0)
```

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

De igual manera, podemos agregarlo en el otro eje

```
np.concatenate((a,b), axis = 1)
```

ValueError: all the input array dimensions **for** the concatenation axis must match exactly, but along dimension 0, the array at index 0 has size 2 **and** the array at index 1 has size 1

Como 'b' es una fila y no una columna, no se puede concatenar a menos que se aplique la transpuesta.

La transpuesta pone nuestro array en sentido opuesto, si el array original es (1,2), con la transpuesta quedará (2,1)

```
b.T
```

```
---> array([[5],  
           [6]])
```

```
np.concatenate((a,b.T), axis = 1)
```

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

- Con la función max puedo obtener el valor más grande de mi array o arreglo
- Con la función min puedo obtener el valor más pequeño de mi array
- Con la función ptp puedo saber cuál es la diferencia entre mi valor más grande y el más pequeño
- Con la función percentil puedo encontrar la mediana de mi array o el porcentaje que yo desee
- Con la función sort puedo organizar mi arreglo de menor a mayor
- Con la función median obtengo la mediana de mi array
- Con la función std puedo encontrar la desviación estándar de mi arreglo
- La desviación estándar al cuadrado es igual a mi varianza
- la media es la suma de todos los valores de mi arreglo, dividido en la cantidad de valores que tengo
- No puedo concatenar 2 arrays de diferentes dimensiones, primero debo modificarlas si es necesario, para que tengan las mismas dimensiones y luego concatenar
- con la T mayúscula creo la transpuesta de una matriz, es decir, cambiar de forma contraria

COPY

`.copy()` nos permite copiar un array de NumPy en otra variable de tal forma que al modificar el nuevo array los cambios no se vean reflejados en array original.

```
arr = np.arange(0, 11)
```

```
arr ----> array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Tomamos un trozo del array original

```
arr[0:6] ----> array([0, 1, 2, 3, 4, 5])
```

```
trozo_de_arr = arr[0:6]
```

Queremos pasar todas nuestras variables a 0

```
trozo_de_arr[:] = 0
```

```
trozo_de_arr ----> array([0, 0, 0, 0, 0, 0])
```

Se han modificado los datos del array original porque seguía haciendo referencia a esa variable.

```
arr ----> array([ 0,  0,  0,  0,  0,  0,  6,  7,  8,  9, 10])
```

Con `.copy()` creamos una copia para no dañar nuestro array original


```
arr_copy = arr.copy()
```

```
arr_copy[:] = 100
```

```
arr_copy ----> array([100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100])
```

```
arr ----> array([ 0,  0,  0,  0,  0,  0,  6,  7,  8,  9, 10])
```

Esta función te ayudará a prevenir muchos errores y tener más confianza a la hora de manipular los datos

 Cuando trabajamos con pedazos de un array padre y empezamos a modificar ese supuesto pedazo, estamos modificando el array padre

```
arr_copy = arr.copy()
```



Aca si tenemos un array que es una copia del padre y podemos modificarlo sin danar el original

...

⚠ Siempre que quieras **modificar** un pedazo del array → COPY

...

- Cuando es por valor, la información de la variable se almacenan en una dirección de memoria diferente al recibirla en la función, por lo tanto si el valor de esa variable cambia no afecta la variable original, solo se modifica dentro del contexto de la función.
- Cuando es por referencia, la variable que se recibe como parámetro en la función apunta exactamente a la misma dirección de memoria que la variable original por lo que si dentro de la función se modifica su valor también se modifica la variable original.



[Enlace de la fuente](#)

CONDICIONES

Las condiciones nos permiten ***hacer consultas más específicas.***

```
arr = np.linspace(1,10,10, dtype = 'int8')
```

```
arr
```

```
---> array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=int8)
```

Regresa un array de booleanos dónde la **condición se cumple.**

```
indices_cond = arr > 5
```

```
indices_cond
```

```
---> array([False, False, False, False, False,  True,  True,  True,  True,  True])
```

Regresa los valores para dónde la condiciones True.

```
arr[indices_cond]
```

```
---> array([ 6, 7, 8, 9, 10], dtype=int8)
```

Múltiples condiciones.

```
arr[(arr > 5) & (arr < 9)]
```

```
---> array([6, 7, 8], dtype=int8)
```

Modificar los valores que cumplan la condición.

```
arr[arr > 5] = 99
```

```
arr
```

```
---> array([ 1, 2, 3, 4, 5, 99, 99, 99, 99, 99], dtype=int8)
```

OPERACIONES

Existen diferentes operaciones que se pueden usar para los arrays de NumPy.

```
lista = [1,2]
```

```
lista ----> [1, 2]
```

Una lista de Python entiende que quieres duplicar los datos. No es lo que buscamos.

```
lista * 2
```

```
---> [1, 2, 1, 2]
```

Pero Numpy lo entiende mucho mejor

```
arr = np.arange(0,10)
```

```
arr2 = arr.copy()
```

```
arr ----> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Ahora multiplicamos por un vector:

```
arr * 2
```

```
---> array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Operación suma de vectores:

```
arr + 2
```

```
---> array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

División con un vector

Como en este caso la primera posición del array es 0, muestra un error pero, no detiene el proceso.

```
1 / arr
```

```
---> RuntimeWarning: divide by zero encountered in true_divide
```

```
"""Entry point for launching an IPython kernel.
```

```
---> array([ inf,  1. , 0.5 , 0.33333333, 0.25 ,0.2, 0.16666667,
0.14285714, 0.125 , 0.11111111])
```

Elevar a un vector:

```
arr**2
```

```
---> array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Sumar dos arrays de igual dimensiones las hace elemento por elemento:

```
arr + arr2
```

```
---> array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Lo mismo aplica para matrices.

```
matriz = arr.reshape(2,5)
```

```
matriz2 = matriz.copy()
```

```
matriz
```

```
---> array([[0, 1, 2, 3, 4],
          [5, 6, 7, 8, 9]])
```

```
matriz - matriz2
```

```
---> array([[0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0]])
```

Una operación importante es la de punto por punto, aquí dos formas de hacerla:

```
np.matmul(matriz, matriz2.T)
```

```
---> array([[ 30,  80],
          [ 80, 255]])
```

```
matriz @ matriz2.T
```

```
---> array([[ 30,  80],
          [ 80, 255]])
```

Para entender por qué se debe utilizar la transpuesta, es importante recordar que el producto punto de dos matrices solo se puede operar si la dimensión de **las columnas** de la primera es igual a la de las **filas** de la segunda.

En el ejemplo:

$$\begin{matrix} (2 \times 5) & & (5 \times 2) \\ & \wedge & \wedge \\ & \text{-----} & \end{matrix}$$

Otro dato es que la matriz resultante queda con dimensiones de las dimensiones que no se usaron, o sea con las filas de la primera y las columnas de la segunda.

Por esto la matriz queda de 2x2.

$$\begin{matrix} (2 \times 5) & & (5 \times 2) \\ & \wedge & \wedge \\ & \text{-----} & \end{matrix}$$

- Es importante realizar las operaciones que deseemos desde las funcionalidades de Numpy, es decir, una lista normal en Python se comporta diferente a un array de Numpy cuando realizamos operaciones
- Existen muchísimas operaciones matemáticas que podemos hacer con nuestros arreglos, debemos practicar
- El producto punto es una operación muy usada en la ciencia de datos

QUIZ NUMPY

1. ¿Qué línea de código genera este array de NumPy?

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

np.arange(0,10)



2. ¿Cómo se asigna de forma correcta al array_copy una copia de el array arr?

array_copy = arr.copy()



3. ¿Con cuál de las siguientes opciones se obtiene el siguiente array de NumPy?

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

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



4. ¿Qué número retorna esta sentencia?

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

2



PANDAS

SERIES Y DATAFRAMES EN PANDAS

Ya entendiste los conceptos básicos de Numpy, ahora hay que entender como funciona la librería de **Pandas**, esta nos ayuda a hacer una mejor exploración y análisis de los datos.

Pandas

Pandas es una librería de Python especializada en el **manejo y análisis de estructuras de datos**. El nombre viene de "Panel data".

- Velocidad
- Poco código
- Múltiples formatos de archivos
- Alineación inteligente

Pandas Series

Es muy parecido a un array de una dimensión (***o vector***) de NumPy.

- Arreglo unidimensional indexado
- Búsqueda por índice
- Slicing
- Operaciones aritméticas
- Distintos tipos de datos

Pandas DataFrame

Muy parecido a las estructuras ***matriciales*** trabajadas con NumPy.

- Estructura principal
- Arreglo de dos dimensiones
- Búsqueda por índice (columnas o filas)
- Slicing
- Operaciones aritméticas
- Distintos tipos de datos
- Tamaño variable

Series

Es un arreglo ***unidimensional*** indexado

```
import pandas as pd
```

Definiendo una lista con ***índices específicos***

```
psg_players = pd.Series(['Navas','Mbappe','Neymar','Messi'],  
index=[1,7,10,30])
```

```
psg_players
```

```
---> 1    Navas  
      7    Mbappe  
     10    Neymar  
     30    Messi  
      dtype: object
```

Búsqueda por ***índices***

```
dict = {1: 'Navas', 7: 'Mbappe', 10: 'Neymar', 30: 'Messi'}
```

```
pd.Series(dict)
```

```
---> 1 Navas
```

```
7 Mbappe
```

```
10 Neymar
```

```
30 Messi
```

```
dtype: object
```

```
psg_players[7]
```

```
----> 'Mbappe'
```

Búsqueda mediante ***Slicing***

```
psg_players[0:3]
```

```
-----> 0    Navas
```

```
         1    Mbappe
```

```
         2    Neymar
```

```
dtype: object
```

Pandas

Similar a la estructura matricial

```
dict = {'Jugador':['Navas','Mbappe','Neymar','Messi'],
```

```
        'Altura':[183.0, 170.0, 170.0, 163.0],
```

```
        'Goles':[2, 200, 150, 500]}
```

```
df_players = pd.DataFrame(dict, index=[1,7,10,30])
```

```
---> Jugador Altura Goles
```

```
     1 Navas    183     2
```

```
     7 Mbappe   170    200
```

```
    10 Neymar   170    150
```

```
    30 Messi    163    500
```

Búsqueda por índices. ***Columnas***


```
df_players.columns
```

```
---> Index(['Jugador', 'Altura', 'Goles'], dtype='object')
```

Búsqueda por **índice**.

```
df_players.index
```

```
-----> RangeIndex(start=0, stop=4, step=1)
```

Reto

Descarga este DataFrame de [Granada FC](#)

- Crea tus propios DataFrames, con los índices que quieras y comparte tus resultados.

PANDAS

Manipulación y análisis de datos. El nombre viene de "Panel data".

- Velocidad
- Poco código
- Múltiples formatos de archivos
- Alineación inteligente

Pandas Series

Es muy parecido a un array de una dimensión (o vector) de NumPy.

- Arreglo unidimensional indexado
- Búsqueda por índice
- Slicing
- Operaciones aritméticas
- Distintos tipos de datos

Pandas DataFrame

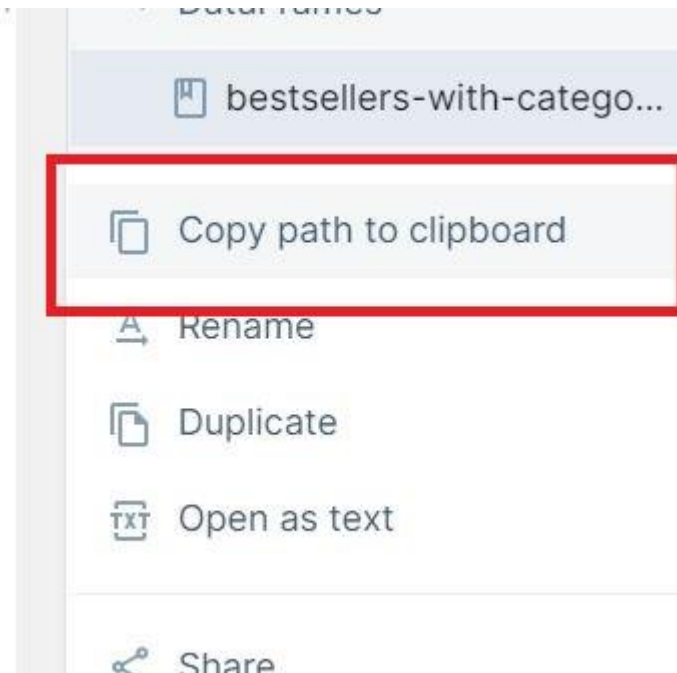
Muy parecido a las estructuras matriciales trabajadas con NumPy.

- Estructura principal
- Arreglo de dos dimensiones
- Búsqueda por índice (columnas o filas)
- Slicing
- Operaciones aritméticas
- Distintos tipos de datos
- Tamaño variable

LEER ARCHIVOS CSV Y JSON CON PANDAS

Para poder leer archivos encontrados en internet (Repositorios, [Kaggle](#)).

Descargamos el [Archivo CSV](#), subimos el archivo a nuestro proyecto, le damos clic derecho sobre él y copiamos la dirección.



import pandas **as** pd


```
pd.read_csv('/work/DataFrames/bestsellers-with-categories.csv')
```

Ejemplo

Niños jugando a los
"Superhéroes"



- En algunas ocasiones el archivo podría estar ***separado por " | "*** y se vería así.

	Name, Author, U... 
	10-Day Gre... 0.2%
	11/22/63: ... 0.2%
	548 others 99.6%
0	10-Day Green Smoothie...
1	11/22/63: A Novel, Stephen...
2	12 Rules for Life: An...
3	1984 (Signet Classics), Geor...
4	5,000 Awesome Facts (About...
5	A Dance with Dragons (A Son...
6	A Game of Thrones / A...
7	A Gentleman in Moscow: A...
8	A Higher Loyalty: Truth...

- Para solucionar esto, usamos el atributo **"Sep = ', ' "** y ya quedará bien organizado.

```
pd.read_csv('/work/DataFrames/bestsellers-with-categories.csv', sep= '
, ')
```

Ejemplo

Niños jugando a los
"Superhéroes"



- Cambiar el **encabezado**, lo podemos hacer con "Header", este pondrá de encabezado los valores que tenga en esa posición.

```
pd.read_csv('/work/DataFrames/bestsellers-with-categories.csv', header  
= 2)
```

	11/22/63: A N... Publicatio... 1.8% StrengthsF... 1.6% 347 others ... 96.5%	Stephen King o... Jeff Kinney _ 2.2% Suzanne Co... _ 2% 245 others _ 95.8%	4.6 float64 3.3 - 4.9	2052 int64 37 - 87841	22 int64 0 - 105
0	12 Rules for Life: An...	Jordan B. Peterson	4.7	18979	15
1	1984 (Signet Classics)	George Orwell	4.7	21424	6
2	5,000 Awesome Facts (About...	National Geographic Kids	4.8	7665	12
3	A Dance with Dragons (A Son...	George R. R. Martin	4.4	12643	11
4	A Game of ...	George R. R.	4.7	19735	30

- **Cambiar el nombre** de las columnas con "names".

```
pd.read_csv('/work/DataFrames/bestsellers-with-categories.csv', header
= 0, names = ['Name', 'Author', 'User Rating', 'Reviews', 'Price',
'Year', 'Genre'])
```

	Name object	Author object	User Rating float	Reviews int64	Price int64
	Publicatio... 1.8%	Jeff Kinney ... 2.2%	3.3 - 4.9	37 - 87841	0 - 105
	StrengthsF... 1.6%	Suzanne Co... 2%			
	349 others ... 96.5%	246 others ... 95.8%			
0	10-Day Green Smoothie...	JJ Smith	4.7	17350	8
1	11/22/63: A Novel	Stephen King	4.6	2052	22
2	12 Rules for Life: An...	Jordan B. Peterson	4.7	18979	15
3	1984 (Signet Classics)	George Orwell	4.7	21424	6
4	5,000 Awesome Facts (About...	National Geographic Kids	4.8	7665	12
5	A Dance with Dragons (A Son...	George R. R. Martin	4.4	12643	11
6	A Game of Thrones / A...	George R. R. Martin	4.7	19735	30

JSON

Para ***agregar un archivo `JSON`***, se hace de igual manera, pero en esta ocasión usamos

```
pd.read_json('/work/DataFrames/hpcharactersdataraw.json')
```

Lo único que cambió en nuestro código fue él 'read_json()'

Reto

- Visita [Kaggle](https://www.kaggle.com/), descarga y carga algún dataset que te llame la atención. Muéstranos que has podido encontrar. 😊

FILTRADO CON LOC Y ILOC

Cuando queremos **navegar** por un dataframe estas funciones permiten filtrar datos de manera más específica

.loc

Filtra según un **label**

```
import pandas as pd
```

```
df_books = pd.read_csv('bestsellers-with-categories.csv', sep=',',  
header=0)
```

```
df_books.loc[:]
```

```
---> #muestra todos los datos del dataframe
```

Mostrar un rango de filas tomando en cuenta el **start y el end**

```
df_books.loc[0:4]
```

```
---> #muestra los datos de la fila 0 a la fila 4
```

- Filtrando por **filas y columnas**

```
df_books.loc[0:4, ['Name', 'Author']]
```

```
----> #filtra los datos de la fila que va de 0 a 4 y de las columnas Name  
y Author
```

- Podemos modificar los valores de una **columna específica** del dataframe

```
df_books.loc[:, ['Reviews']] * -1
```

```
---> #multiplica por -1 todos los valores de la columna Reviews
```

- Filtrar datos que cumplan una **condición** determinada

```
df_books.loc[:, ['Author']] == 'JJ Smith'
```

```
----> #muestra la columna Author con True en los valores que cumplen  
la condicion y False para los que no la cumplen
```

.iloc

Filtra mediante **índices**.

```
df_books.iloc[:] ---> #muestra todos los datos del dataframe
```

- Filtrar datos según los índices de las **filas y las columnas**

`df_books.iloc[:4, 0:2]` ---> #muestra los datos de las filas que van de 0 a 3 y las columnas con índices 0 y 1

- Buscar un **dato específico**.

`df_books.iloc[1,3]` ---> #muestra el dato alojado en la fila 1 columna 3

- Tenemos dos atributos que me permiten navegar dentro de mis DataFrame y ellos son: `iloc` y `loc`
- Para filtrar por columnas escribo el nombre de la columna que deseo filtrar de la siguiente manera:
`Nombre_del_df["Nombre_De_columna"]`
- Para filtrar por columna y filas al mismo tiempo lo hago con la siguiente sentencia: `Nombre_del_df[start : stop,["Nombre_De_columna"]]`
- Puedo filtrar y además hacer operaciones con lo que desee
- Con `loc` puedo filtrar simultáneamente filas y columnas
- Con `iloc` puedo realizar filtrados entrando directamente al índice de las columnas a diferencia de `loc`, que debo escribir el nombre
- `loc` es a través de labels e `iloc` es a través de índices

Python Pandas Selections and Indexing

.iloc selections - position based selection

`data.iloc[<row selection>, <column selection>]`

Integer list of rows: [0,1,2]

Slice of rows: [4:7]

Single values: 1

Integer list of columns: [0,1,2]

Slice of columns: [4:7]

Single column selections: 1

loc selections - position based selection

`data.loc[<row selection>, <column selection>]`

Index/Label value: 'john'

List of labels: ['john', 'sarah']

Logical/Boolean index: data['age'] == 10

Named column: 'first_name'

List of column names: ['first_name', 'age']

Slice of columns: 'first_name':'address'



AGREGAR O ELIMINAR DATOS CON PANDAS

Muchas ocasiones necesitamos agregar, eliminar o separar datos y pandas nos ofrece varias funciones para que este proceso se vuelva mucho más sencillo.

- Muestra las primeras 5 líneas del DataFrame

```
df_books.head()
```

---> muestra las primeras 5 lineas del dataframe

- Eliminar columnas de la salida pero no del DataFrame

```
df_books.drop('Genre', axis=1).head()
```

---> #elimina la columna Genre de la salida pero no del dataframe

- Eliminar una columna

```
del df_books['Price']
```

---> #elimina la columna Price del dataframe

- Eliminar filas

```
df_books.drop(0, axis=0)
```

---> #elimina la fila 0 del dataframe

- Eliminar un conjunto de filas mediante una lista

```
df_books.drop([0,1,2], axis=0)
```

---> #elimina las filas 0, 1 y 2 del dataframe

- Elimina un conjunto de filas mediante un rango

```
df_books.drop(range(0,10), axis=0)
```

---> #elimina las primeras 10 filas del dataframe

- Agregar una nueva columna con valores Nan

```
df_books['Nueva_columna'] = np.nan
```

---> #Crea una nueva columna con el nombre de Nueva_columna de valores Nan

- Mostrar el número de filas o columnas que tiene un DataFrame

```
df_books.shape[0]
```

```
---> #Muestra el numero de filas que posee el dataframe
```

- Agregar valores a una nueva columna

```
data = np.arange(0, df_books.shape[0])
```

- Crear una nueva columna y agregar los valores almacenados en el array

```
df_books['Rango'] = data
```

```
---> #Crea una nueva columna llamada Rango con los valores del array
```

- Para añadir filas se utiliza la función `append` de Python añadiendo como parámetro una lista, diccionario o añadiendo los valores manualmente.

```
df_books.append(df_books)
```

```
---> #Duplica las filas del dataframe porque se agrega a si mismo
```

MANEJO DE DATOS NULOS

Los datos nulos ***son dolores de cabeza*** para este mundo de la ciencia de datos y se van a encontrar mucho en nuestros DataFrames

- Creamos un DataFrame con algunos valores nulos

```
import pandas as pd
```

```
import numpy as np
```

```
dict = {'Col1':[1,2,3,np.nan],
```

```
'Col2':[4, np.nan,6,7],
```

```
'Col3':['a','b','c', None]}
```

```
df = pd.DataFrame(dict)
```

```
---> Col1 Col2 Col3
```

```
0  1     4   a
```

```
1  2    nan  b
```

```
2  3    6  c
```

```
3 nan    7 None
```

- **Identificar** valores nulos en un DataFrame

```
df.isnull()
```

```
----> Col1 Col2 Col3
```

```
0    false false false
```

```
1    false true  false
```

```
2    false false false
```

```
3    true  false true
```

- Identificar valores nulos con un valor **numérico**

```
df.isnull()*1
```

```
---> Col1 Col2 Col3
```

```
0    0    0    0
```

```
1    0    1    0
```

```
2    0    0    0
```

```
3    1    0    1
```

- Sustituir los valores nulos **por una cadena**

```
df.fillna('Missing')
```

```
---> Col1 Col2 Col3
```

```
0    1.0  4.0  a
```

```
1    2.0 Missing b
```

```
2    3.0  6.0  c
```

```
3    Missing 7.0 Missing
```

- Sustituir valores nulos por una **medida estadística** realizada con los valores de las columnas

```
df.fillna(df.mean())
```

```
----> Col1 Col2 Col3
```

```
0    1    4  a
```

1	2	5.667	b
2	3	6	c
3	2	7	None

- Sustituir valores nulos por valores de **interpolación**

df.interpolate()

```
----> Col1 Col2 Col3
0      1    4    a
1      2    5    b
2      3    6    c
3      3    7   None
```

- **Eliminar** valores nulos

df.dropna()

```
---> Col1 Col2 Col3
0     1    4    a
2     3    6    c
```

NaN, None y NaT:

- **NaN:** si una columna es de tipo numérico y falta algún valor, ese valor será NaN (Not a Number). Como curiosidad, NaN es de tipo float y, por tanto, si tienes una columna de enteros y hay un valor que falta, automáticamente toda esa columna pasa a ser de tipo float debido al NaN (se hace upcasting a cada valor).
- **NaT:** si tienes una columna de tipo DateTime y falta algún valor, ese será NaT (Not a Time).
- **None:** cuando tenemos una columna de tipo object (el tipo de los strings). Aunque para estas columnas podríamos encontrar cualquiera de los 3: None, NaN y NaT.

Nota aparte: np.NaN == np.NaN devuelve False, al igual que pd.NaT == pd.NaT devuelve False. Sin embargo, ``None == None``

FILTRADO POR CONDICIONES

Funciona por lógica booleana y retorna los valores que están en "True". Es muy útil porque en ocasiones queremos filtrar o separar datos.

- Llamamos los datos de un archivo csv para manejarlos

```
df_books = pd.read_csv('bestsellers-with-categories.csv')
```

```
df_books.head(2) ---> #muestra los primeros dos registros del  
dataFrame
```

- Mostrar datos que sean **mayores** a cierto valor

```
mayor2016 = df_books['Year'] > 2016
```

```
mayor2016
```

```
---> #muestra el dataFrame con valores booleanos. True para libros  
publicados desde el 2017
```

- Filtrar datos **en nuestro DataFrame** que sean mayores a cierto valor

```
df_books[mayor2016]
```

```
---> #filtra los datos que cumplen con la condicion
```

- También se puede colocar la **condición directamente** como parámetro

```
df_books[df_books['Year'] > 2016]
```

```
---> #filtra los datos que cumplen con la condicion
```

- Mostrar los datos que sean **igual** a cierto valor

```
genreFiction = df_books['Genre'] == 'Fiction'
```

```
genreFiction ---> #muestra el dataFrame con valores booleanos. True  
para libros de tipo Fiction
```

- Filtrado con **varias condiciones**

```
df_books[genreFiction & mayor2016]
```

```
---> #Filtra los libros que sean de tipo Fiction y que hayan sido  
publicado desde 2017
```

- Filtrado con **negación**

```
df_books[~mayor2016]
```

---> #Filtra los libros publicados antes o igual al 2016

Símbolos de condicionales con Pandas y NumPy

Símbolo	Equivalencia	Ejemplo
&	AND	(a>b & c>d)
	OR	(a>b or c>d)
	OR	
	OR	
¬	NOT	not(a>b)
~	NOT	
~	NOT	

- El símbolo que va encima de la ñ en español se llama virgulilla
- Si nuestro teclado es diferente al español o no nos sale la virgulilla presionando las teclas **<ALT Gr> + 4**, podemos utilizar las siguientes combinaciones de teclas.
- En **Windows** (si no es teclado español) se hace con la combinación de las teclas **<Alt> + 126**
- En **Linux** (si no es teclado español) se hace con la combinación de las teclas **<Alt> + ñ**, si es teclado español igual que en Windows
- En **Mac** (si es teclado español) se hace con la combinación de las teclas **<Alt> + ñ**, sino se debe colocar manualmente el símbolo, a través de la paleta de caracteres
- Pandas también nos permite filtrar por condiciones
- Puedo crear una condición y luego decirle a mi DataFrame que me muestre los datos cumpliendo esta condición
- Puedo filtrar por dos o más condiciones utilizando este carácter &
- En pandas con este carácter ~ puedo negar una condición

FUNCIONES PRINCIPALES DE PANDAS

Hay ciertas **funciones** que son muy importantes y que siempre estaremos usando a la hora de hacer análisis de datos, para mayor facilidad y comprensión del DataFrame.

- **Mostrar** las primeras dos líneas de registro

```
df_books.head(2)
```

```
---> #muestra los primeros dos registros del dataframe
```

- Mostrar **los diferentes datos** que contiene el DataFrame

```
df_books.info()
```

```
---> py
```

```
RangeIndex: 550 entries, 0 to 549      #numero de registro
```

```
Data columns (total 7 columns):      #total de columnas
```

#	Column	Non-Null Count	Dtype	#tipos de cada columna
0	Name	550 non-null	object	
1	Author	550 non-null	object	
2	User Rating	550 non-null	float64	
3	Reviews	550 non-null	int64	
4	Price	550 non-null	int64	
5	Year	550 non-null	int64	
6	Genre	550 non-null	object	

```
dtypes: float64(1), int64(3), object(3)
```

- Obtener diferentes **datos estadísticos** de las columnas numéricas.

```
df_books.describe()
```

```
---> User.Rating  Reviews  Price  Year
```


count	550	550	550	550
mean	4.618	11953.281	13.1	2014
std	0.226	11731.132	10.84	3.165
min	3.3	37	0	2009
25%	4.5	4058	7	2011
50%	4.7	8580	11	2014
75%	4.8	17253.25	16	2017
max	4.9	87841	105	2019

- Mostrar los **últimos 5 registros** del DataFrame

```
df_books.tail()
```

```
---> #muestra los ultimos 5 registros
```

- Obtener el **uso de la memoria** de cada columna

```
df_books.memory_usage(deep=True)
```

```
--->
```

Index	128
Name	59737
Author	39078
User Rating	4400
Reviews	4400
Price	4400
Year	4400
Genre	36440

```
dtype: int64
```

- Obtener **cuantos datos** tenemos de algo en específico

```
df_books['Author'].value_counts()
```

```
---> Muestra cuantos datos hay de cada autor
```

- **Eliminar** registros duplicados

```
df_books.drop_duplicates()
```

- Ordenar los **registros según valores** de la columna (orden ascendente)

```
df_books.sort_values('Year')
```

---> #ordena los valores de menor a mayor segun el año

- Ordenar los registros según valores de la columna (**orden descendente**)

```
df_books.sort_values('Year', ascending=False)
```

---> #ordena los valores de mayor a menor segun el año

Reto

En este artículo de [Pandas](#) podrás encontrar las funciones más usadas

- Carga un DataSet de tu preferencia e implementa estas funciones y cuéntanos cuál te ha parecido más interesante

✓ **.head()** → trae los primeros datos

✓ **.info()** → Columnas, indices, cuales noson nulos, tipo de dato que maneja

✓ **.describe()** → Solo de las columnas numericas me arroja datos estadisticos [media,max,miun,mediana,etc]

✓ **.memory_usage()** → memoria utilizada

✓ **.value_counts()** → cuenta valores de una columna

✓ **.drop_duplicates()** → elimina los valores repetidos

✓ **.sort_values(columna para ordenar)** → Se puede ordenar de forma descendiente con la bandera **ascending=False**

GROUPBY

Permite **agrupar datos en función de los demás**. Es decir, hacer el análisis del DataFrame en función de una de las columnas.

- Agrupar por Author y mostrar el **conteo** de los datos de las demás columnas

```
df_books.groupby('Author').count()
```

```
--->
```

	Name	User	Rating	Reviews	Price	Year	Genre
Abraham Verghese	2	2	2	2	2	2	
Adam Gasiewski	1	1	1	1	1	1	
Adam Mansbach	1	1	1	1	1	1	
Adir Levy	1	1	1	1	1	1	

- Agrupar por Author y mostrar la **media** de los datos de las demás columnas

```
df_books.groupby('Author').median()
```

```
--->
```

	User Rating	Reviews	Price	Year
Abraham Verghese	4.6	4866	11	2010.5
Adam Gasiewski	4.4	3113	6	2017
Adam Mansbach	4.8	9568	9	2011
Adir Levy	4.8	8170	13	2019

La columna Author, en los casos anteriores, pasa a ser el índice.

- Podemos usar loc y acceder a un **dato específico** del DataFrame. Agrupar por autor y mostrar la suma de los valores de las demás columnas para William Davis

```
df_books.groupby('Author').sum().loc['William Davis']
```

```
--->
```

User Rating	8.8
Reviews	14994.0
Price	12.0

Year 4025.0

Name: William Davis, dtype: float64

- Agrupar por author y mostrar la suma de los valores de las demás columnas. Colocar los índices que el DataFrame trae por defecto

```
df_books.groupby('Author').sum().reset_index()
```

```
--->
   Author  User Rating  Reviews  Price  Year
0  Abraham Verghese    9.2    9732    22  4021
1   Adam Gasiewski    4.4    3113     6  2017
2   Adam Mansbach    4.8    9568     9  2011
3   Adir Levy       4.8    8170    13  2019
```

- La función `agg()` permite aplicar varias funciones al DataFrame una vez agrupado según una columna específica. Agrupar por Author y mostrar el mínimo y máximo de las demás columnas

```
df_books.groupby('Author').agg(['min','max'])
```

---> #muestra cada columna dividida en dos: min y max. Estas contienen los valores maximo y minimo de la columna para cada Author

- Agrupar por Author, obtener **el mínimo y máximo** de la columna 'Reviews' y sumar los valores de la columna 'User Rating'

```
df_books.groupby('Author').agg({'Reviews':['min','max'], 'User Rating':'sum'})
```

```
--->
   Reviews min  Reviews max  User Rating
Abraham Verghese    4866    4866         9.2
Adam Gasiewski     3113    3113         4.4
Adam Mansbach      9568    9568         4.8
Adir Levy          8170    8170         4.8
```

- Agrupar por 'Author - Year' y contar los valores de las demás columnas

```
df_books.groupby(['Author','Year']).count()
```

```
--->
   Name  User Rating  Reviews  Price  Genre
('Abraham Verghese', 2010)  1     1     1     1     1
```

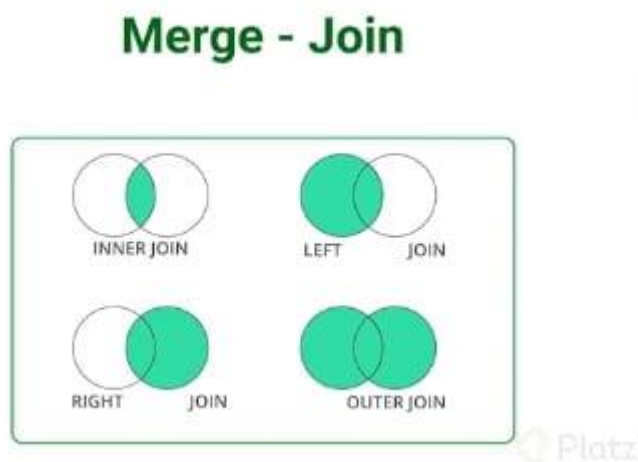
('Abraham Verghese', 2011)	1	1	1	1	1
('Adam Gasiewski', 2017)	1	1	1	1	1
('Adam Mansbach', 2011)	1	1	1	1	1

Reto

Lee este [artículo sobre el método groupby](#) y cuéntanos que otras funciones de agregación podemos usar

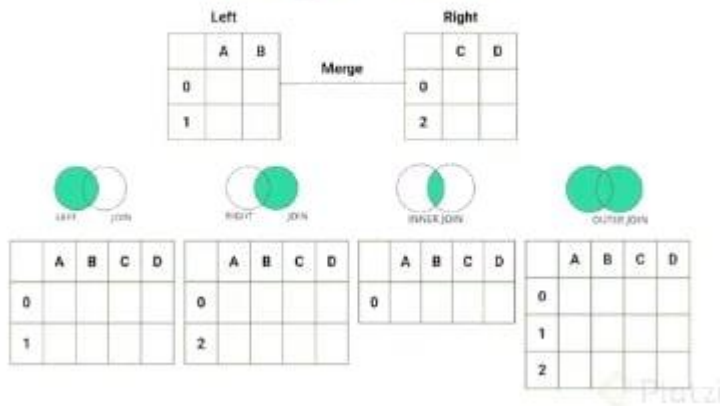
COMBINANDO DATAFRAMES

Existen diferentes formas de fusionar dos DataFrames. Esto se hace a través de la **lógica de combinación** como se muestra a continuación:



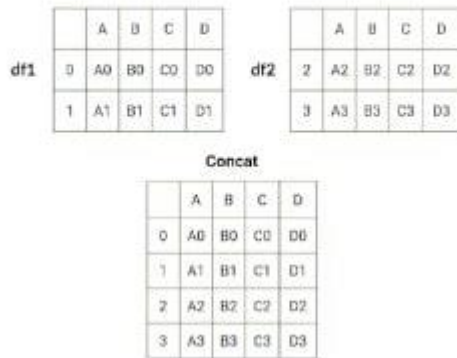
- **Left join:** Da prioridad al DataFrame de la izquierda. Trae siempre los datos de la izquierda y las filas en común con el DataFrame de la derecha.
- **Right join:** Da prioridad al DataFrame de la derecha. Trae siempre los datos de la derecha y las filas en común con el DataFrame de la izquierda.
- **Inner join:** Trae solamente aquellos datos que son común en ambos DataFrame
- **Outer join:** Trae los datos tanto del DataFrame de la izquierda como el de la derecha, incluyendo los datos que comparten ambos.

Merge - Join



- Concat - Axis 0: permite combinar dos dataframes a nivel de filas. Crecimiento vertical

Concat - Axis 0



- Concat - Axis 1: permite combinar dos dataframes a nivel de columnas. La organizacion por columnas no va a ser la misma para ambos dataFrames, por tanto, se crearan valores NaN para rellenar los espacios vacios. Crecimiento horizontal

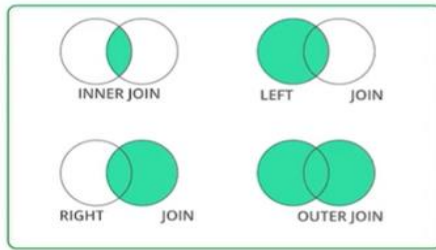
Concat - Axis 1

df1		A	B	C	D	df2		C	D	E	F
	0	A0	B0	C0	D0		1	C1	D1	E1	F1
	1	A1	B1	C1	D1		2	C2	D2	E2	F2

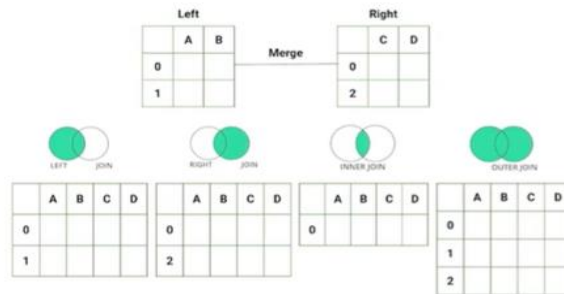
Concat											
	A	B	C	D	C	D	E	F			
0	A0	B0	C0	D0	NaN	NaN	NaN	NaN			
1	A1	B1	C1	D1	C1	D1	E1	F1			
2	NaN	NaN	NaN	NaN	C2	D2	E2	F2			



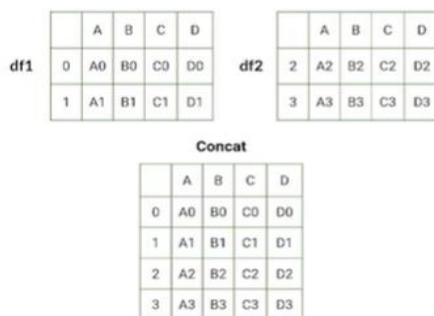
Merge - Join



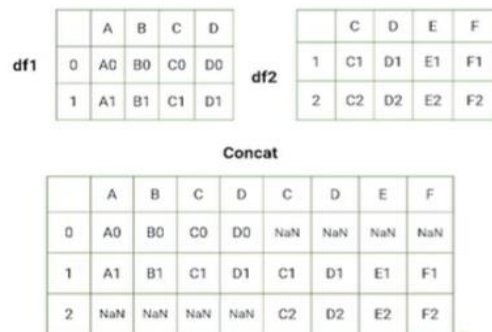
Merge - Join



Concat - Axis 0



Concat - Axis 1



- Cuando combinamos 2 conjuntos de datos y utilizamos left join, esto quiere decir, que me traerá todos los datos que existan en el conjunto de la izquierda junto con la intersección entre los dos conjuntos
- Cuando utilizamos right join, la prioridad está en el conjunto de la derecha, es decir, nos traerá todos los datos del conjunto de la derecha y los que se encuentren en la intersección de los conjuntos
- Cuando utilizamos inner join solamente traerá los datos que existan en ambos conjuntos de datos
- Cuando utilizamos outer join nos va a traer todos los datos que existan en los conjuntos de datos
- En pandas utilizamos la función concat para unir algunos DataFrame, es importante saber que con esta función debemos definir los Axis
- Si utilizamos el Axis=0 fusionamos por filas, si utilizamos el Axis=1 la fusión se realizará por columnas

MERGE Y CONCAT

Como podemos usar la lógica anteriormente vista en código, usando los parámetros de Pandas

- Importamos Pandas y Numpy

```
import pandas as pd
```

```
import numpy as np
```

Concat

- En esta ocasión vamos a crear un DataFrame nuevo

```
df1 = pd.DataFrame({'A':['A0', 'A1', 'A2','A3'],  
                    'B':['B0', 'B1', 'B2','B3'],  
                    'C':['C0', 'C1', 'C2','C3'],  
                    'D':['D0', 'D1', 'D2','D3']})
```

```
df2 = pd.DataFrame({'A':['A4', 'A5', 'A6','A7'],  
                    'B':['B4', 'B5', 'B6','B7'],  
                    'C':['C4', 'C5', 'C6','C7'],  
                    'D':['D4', 'D5', 'D6','D7']})
```

- **Concatenar** los DataFrames

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

```
---> A  B  C  D
```

```
0  A0  B0  C0  D0
```

```
1  A1  B1  C1  D1
```

```
2  A2  B2  C2  D2
```

```
3  A3  B3  C3  D3
```

```
0  A4  B4  C4  D4
```

```
1  A5  B5  C5  D5
```

2 A6 B6 C6 D6

3 A7 B7 C7 D7

- **Corregir** los índices

```
pd.concat([df1,df2], ignore_index= True)
```

---> A B C D

0 A0 B0 C0 D0

1 A1 B1 C1 D1

2 A2 B2 C2 D2

3 A3 B3 C3 D3

4 A4 B4 C4 D4

5 A5 B5 C5 D5

6 A6 B6 C6 D6

7 A7 B7 C7 D7

- Por **axis 1**

```
pd.concat([df1,df2], axis = 1)
```

---> A B C D A.1 B.1 C.1 D.1

0 A0 B0 C0 D0 A4 B4 C4 D4

1 A1 B1 C1 D1 A5 B5 C5 D5

2 A2 B2 C2 D2 A6 B6 C6 D6

3 A3 B3 C3 D3 A7 B7 C7 D7

Merge

- Creamos DataFrame

```
izq = pd.DataFrame({'key' : ['k0', 'k1', 'k2','k3'],  
  'A' : ['A0', 'A1', 'A2','A3'],  
  'B': ['B0', 'B1', 'B2','B3']})
```

```
der = pd.DataFrame({'key' : ['k0', 'k1', 'k2','k3'],
```

```
'C' : ['C0', 'C1', 'C2','C3'],  
'D': ['D0', 'D1', 'D2','D3']})
```

- **Unir** el DataFrame Der a Izq

```
izq.merge(der)
```

```
---> key A  B  C  D
```

```
0  k0  A0  B0  C0  D0
```

```
1  k1  A1  B1  C1  D1
```

```
2  k2  A2  B2  C2  D2
```

```
3  k3  A3  B3  C3  D3
```

MERGE 2

```
izq = pd.DataFrame({'key' : ['k0', 'k1', 'k2','k3'],  
  'A' : ['A0', 'A1', 'A2','A3'],  
  'B': ['B0', 'B1', 'B2','B3']})
```

```
der = pd.DataFrame({'key_2' : ['k0', 'k1', 'k2','k3'],  
  'C' : ['C0', 'C1', 'C2','C3'],  
  'D': ['D0', 'D1', 'D2','D3']})
```

- Hay diferencias entre algunas columnas, por esa razón hay que **separarlos** de esta manera:

```
izq.merge(der, left_on = 'key', right_on='key_2')
```

```
---> key A  B  key_2  C  D
```

```
0  k0  A0  B0  k0   C0  D0
```

```
1  k1  A1  B1  k1   C1  D1
```

```
2  k2  A2  B2  k2   C2  D2
```

```
3  k3  A3  B3  k3   C3  D3
```

MERGE 3

```
izq = pd.DataFrame({'key' : ['k0', 'k1', 'k2','k3'],
```

```
'A' : ['A0', 'A1', 'A2','A3'],  
'B': ['B0', 'B1', 'B2','B3']})
```

```
der = pd.DataFrame({'key_2' : ['k0', 'k1', 'k2',np.nan],  
 'C' : ['C0', 'C1', 'C2','C3'],  
'D': ['D0', 'D1', 'D2','D3']})
```

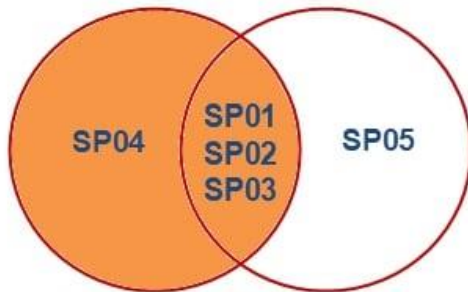
- Si tenemos un NaN en nuestro DataFrame, pandas **no lo detectará como un mach**. Se soluciona con How, dando así, una preferencia.

```
izq.merge(der, left_on = 'key', right_on='key_2', how='left')
```

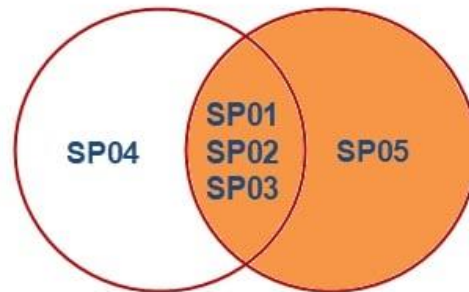
```
---> key A B key_2 C D  
0 k0 A0 B0 k0 C0 D0  
1 k1 A1 B1 k1 C1 D1  
2 k2 A2 B2 k2 C2 D2  
3 k3 A3 B3 NaN NaN NaN
```

- Utilizando la función concat puedo unir diferentes DataFrame y estos deben ir dentro de []
- Por defecto la fusión se hace por el Axis = 0, es decir por las filas, si utilizamos el Axis = 1, la fusión se realizará por las columnas
- También podemos utilizar la función merge para fusionar 2 DataFrame, lo ideal seria que nuestros DataFrame tengan una llave en común para realizar un correcto merge
- Cuando los conjuntos de datos no poseen una llave en común, debemos especificar que llave se usara en right y que llave en left
- Utilizando el parámetro how, puedo definir qué tipo de unión voy a realizar. Muy parecido a las consultas que se hacen en SQL

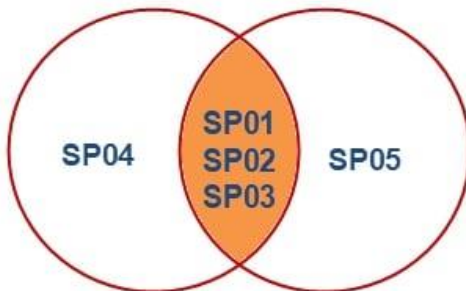
PANDAS - MERGE



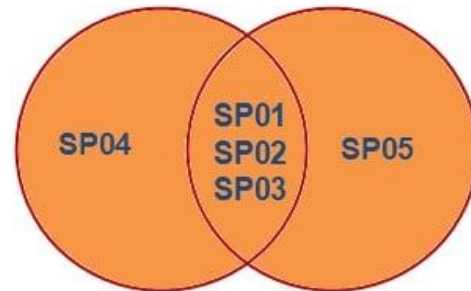
LEFT



RIGHT



INNER



OUTER

JOIN

Join Es otra herramienta para hacer exactamente lo mismo, una combinación. La diferencia es que **join va a ir a los índices y no a columnas específicas.**

```
izq = pd.DataFrame({'A': ['A0','A1','A2'],  
                    'B':['B0','B1','B2']},  
                    index=['k0','k1','k2'])
```

```
der =pd.DataFrame({'C': ['C0','C1','C2'],  
                   'D':['D0','D1','D2']},  
                   index=['k0','k2','k3'])
```

- Combinamos izq con der

```
izq.join(der)
```

```
---> A  B  C  D
```

```
k0  A0  B0  C0  D0
```

```
k1  A1  B1  nan nan
```

```
k2  A2  B2  C1  D1
```

- Traer todos los datos aunque no hagan match.

```
izq.join(der, how = 'outer')
```

```
---> A  B  C  D
```

```
k0  A0  B0  C0  D0
```

```
k1  A1  B1  nan nan
```

```
k2  A2  B2  C1  D1
```

```
k3  nan nan C2  D2
```

PIVOT Y MELT

Pivot y Melt

22/24

Ir a la nueva versión

LECTURA

Hola, te doy la bienvenida a la clase de **pivot_table** y **melt**, dos funciones que sirven para cambiar la estructura de nuestro DataFrame de acuerdo a nuestras necesidades.

pivot_table

Esta función puede traer recuerdos a las personas interesadas en el mundo del SQL, ya que Oracle, PostgreSQL y otros motores de bases de datos la tienen implementada desde hace muchos años. Pivot, básicamente, transforma los valores de determinadas columnas o filas en los índices de un nuevo DataFrame, y la intersección de estos es el valor resultante.

Entiendo que esto puede sonar algo confuso, pero no te preocupes, todo queda mucho más claro con un ejemplo.

1. Para comenzar, crea un nuevo Jupyter Notebooks, puedes usar Google Colab o la notebook de tu preferencia que estés utilizando para este curso.
2. Carga el DataFrame que hemos usado en el curso:

```
df_books = pd.read_csv('bestsellers with  
categories.csv',sep=',',header=0)
```

3. Explóralo viendo sus primeras 5 filas:

```
df_books.head()
```

4. Aplica pivot_table:

```
df_books.pivot_table(index='Author',columns='Genre',values='User  
Rating')
```

	Name	Author	User Rating	Reviews	Price	Year	Genre
0	10-Day Green Smoothie Cleanse	JJ Smith	4.7	17350	8	2016	Non Fiction
1	11/22/63: A Novel	Stephen King	4.6	2052	22	2011	Fiction
2	12 Rules for Life: An Antidote to Chaos	Jordan B. Peterson	4.7	18979	15	2018	Non Fiction
3	1984 (Signet Classics)	George Orwell	4.7	21424	6	2017	Fiction
4	5,000 Awesome Facts (About Everything!) (Nati...	National Geographic Kids	4.8	7665	12	2019	Non Fiction
5	A Dance with Dragons (A Song of Ice and Fire)	George R. R. Martin	4.4	12643	11	2011	Fiction

Como resultado, los valores de Author pasan a formar el índice por fila y los valores de Genre pasan a formar parte de los índices por columna, y el User Rating se mantiene como valor.

Genre		Fiction	Non Fiction
Author			
Abraham Verghese		4.6	NaN
Adam Gasiewski		NaN	4.4
Adam Mansbach		4.8	NaN
Adir Levy		4.8	NaN
Admiral William H. McRaven		NaN	4.7
Adult Coloring Book Designs		NaN	4.5

248 rows x 2 columns [Open in new tab](#)

Por supuesto, para este caso, un Author suele tener un solo género literario, así que no es una transformación muy útil, pero veamos si podemos lograr algo mejor.

5. Ejecuta la siguiente variación:

```
df_books.pivot_table(index='Genre',columns='Year', values='User Rating',aggfunc='sum')
```

En este caso tenemos por cada género, la suma a lo largo de los años. Esto es mucho más interesante, ¿verdad? La mejor noticia es que no solo podemos obtener la suma, también podemos obtener la media, la desviación estándar, el conteo, la varianza, etc. Únicamente con cambiar el parámetro aggfunc que traduce función de agrupamiento.

Year Genre	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
Fiction	110.2	92.3	97.0	94.4	109.1	134.3	79.1	89.6	113.7	99.5	96.4
Non Fiction	119.0	135.6	130.9	132.2	118.6	96.8	153.3	144.3	119.3	133.9	140.6

melt

El método melt toma las columnas del DataFrame y las pasa a filas, con dos nuevas columnas para especificar la antigua columna y el valor que traía.

Por ejemplo, simplemente al imprimir las cinco primeras filas del DataFrame con las columnas de Name y Genre se tiene este resultado.

1. Para ello ejecuta la siguiente línea en tu Jupyter Notebook:

```
df_books[['Name','Genre']].head(5)
```

	Name	Genre
0	10-Day Green Smoothie Cleanse	Non Fiction
1	11/22/63: A Novel	Fiction
2	12 Rules for Life: An Antidote to Chaos	Non Fiction
3	1984 (Signet Classics)	Fiction
4	5,000 Awesome Facts (About Everything!) (Nati...	Non Fiction

2. Aplica melt de la siguiente manera:

```
df_books[['Name','Genre']].head(5).melt()
```

	variable	value
0	Name	10-Day Green Smoothie Cleanse
1	Name	11/22/63: A Novel
2	Name	12 Rules for Life: An Antidote to Chaos
3	Name	1984 (Signet Classics)
4	Name	5,000 Awesome Facts (About Everything!) (Nati...
5	Genre	Non Fiction
6	Genre	Fiction
7	Genre	Non Fiction
8	Genre	Fiction
9	Genre	Non Fiction

Ahora cada resultado de las dos columnas pasa a una fila de este modo a tipo **llave:valor**.

3. En el siguiente ejemplo ejecutemos melt de esta manera:

```
df_books.melt(id_vars='Year',value_vars='Genre')
```

	Year	variable	value
0	2016	Genre	Non Fiction
1	2011	Genre	Fiction
2	2018	Genre	Non Fiction
3	2017	Genre	Fiction
4	2019	Genre	Non Fiction
5	2011	Genre	Fiction

Simplemente, podemos seleccionar las columnas que no quiero hacer melt usando el parámetro `id_vars`. Para este caso Year y también la única columna que quiero aplicar el melt, para este caso Genre con la propiedad `value_vars`.

Hemos conocido pivot y melt, dos herramientas muy útiles para manipular nuestros DataFrames.

APPLY

Apply Es un comando muy poderoso que nos deja aplicar funciones a nuestro DataFrame

- Creamos un DataFrame habitual

```
import pandas as pd
```

```
df_books = pd.read_csv('/work/DataFrames/bestsellers-with-categories.csv')
```

```
df_books.head(2)
```

- Creamos nuestra **función**

```
def two_times(value):
```

```
    return value * 2
```

- Lo aplicamos a la columna de User Rating

```
df_books['User Rating'].apply(two_times)
```

---> Se multiplica por 2 todos los valores de la columna

- Podemos guardarlo en una columna nueva

```
df_books['User Rating2'] =df_books['User Rating'].apply(two_times)
```

- Se pueden crear **lambda functions**

```
df_books['User Rating2'] =df_books['User Rating'].apply(lambda x: x*3)
```

---> Multiplica todos los valores por 3

- Apply en **varias columnas con condiciones**, hay que especificar a que los vamos a aplicar (filas o columnas)

```
df_books.apply(lambda x: x['User Rating'] * 2 if x['Genre'] == 'Fiction'
else x['User Rating'], axis = 1)
```

---> Multiplica por 2 a los datos que cumplan la condición

CIERRE

POSIBILIDADES CON PANDAS Y NUMPY

Numpy

- Que es, que bondades tiene, porque se usa
- Crear arrays
- hacer slicing
- filtrar datos
- aplicar funciones aritméticas

Pandas

- Que son Series- DataFrames
- Como llevar diferentes archivos
- Iloc y loc
- Apply
- Merge, concat y join

Lo que aprendimos es lo que se **usa cotidianamente** en la ciencia de datos, esto nos ayudará a aplicar a computer vision con redes neuronales convolucionales (puedes aprender más de esto en el [Curso Profesional de Computer Vision con TensorFlow](#)), manipular, hacer visualizaciones.

En sí, **este es el principio** de un gran mundo que nos espera para seguir aprendiendo y creciendo sin igual.