😎 Análisis y Manipulación de datos con Pandas

Pandas es una biblioteca de Python de código abierto que proporciona estructuras de datos y herramientas de análisis de datos de alto rendimiento y fáciles de usar para el lenguaje de programación Python. Con Pandas, podemos lograr cinco pasos típicos en el procesamiento y análisis de datos, independientemente del origen de los datos: cargar, preparar, manipular, modelar y analizar.

pip install pandas

Estructura de Datos	Dimensiones	Descripción
Serie	1	Matriz homogénea etiquetada 1D, tamaño inmutable
Dataframe	2	Estructura tabular etiquetada en 2D, de tamaño variable con columnas de tipos heterogéneos

🧐 Series

Serie: Es una matriz unidimensional como estructura con datos homogéneos. Por ejemplo:

10	23	56	17	52	61	73	90	26	72

- **Se puede crear una serie de pandas utilizando el siguiente constructor: pandas.Series (data, index, dtype, copy)**
- **Se puede crear una serie usando varias entradas como: narray (array de numpy) , dict, valor escalar o constante.**

In [1]:
```
import pandas as pd
```

Crear una serie vacía:

In [2]:
```
s = pd.Series(dtype=float)
s
```

Out[2]:
```
Series([], dtype: float64)
```

Crear una serie desde escalar

Si los datos son un valor escalar, se debe proporcionar un índice. El valor se repetirá para que coincida con la longitud del índice:

In [3]:
```
s = pd.Series(8, index=[0, 1, 2, 3])
s
```

Out[3]:
```
0    8
1    8
2    8
3    8
dtype: int64
```

Crear una serie desde un lista:

In [4]:
```
data = ['a','b','c','d']
s = pd.Series(data)
s
```

Out[4]:
```
0    a
1    b
2    c
3    d
dtype: object
```

La serie tiene una secuencia de valores y una secuencia de índices a los que podemos acceder con los valores y los atributos de índice. Aquí no pasamos ningún índice, por lo que, por defecto, asignó los índices que van desde 0 a len(s) -1 es decir, 0 a 3

Crear una serie desde diccionario:

Si no se especifica ningún índice, las claves del diccionario se toman para construir el índice. Si se pasa el índice, se extraerán los valores en los datos correspondientes a las etiquetas del índice.

In [5]:
```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
s
```

Out[5]:
```
a    0.0
b    1.0
c    2.0
dtype: float64
```

Observa: las claves del diccionario se utilizan para construir el índice.

In [6]:
```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data,index=['b','c','d','a'])
s
```

Out[6]:
```
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Observa: el orden del índice persiste y el elemento que falta se llena con NaN.

crear una serie con numpy

In [7]:
```
import numpy as np

s = pd.Series(np.random.randn(4))
s
```

Out[7]:
```
0   -0.846238
1   -0.038588
2   -0.487022
3   -0.743002
dtype: float64
```

La diferencia esencial entre pandas y numpy es el índice:

- **La matriz NumPy tiene un índice entero definido implícitamente que se usa para acceder a los valores.**
- **La serie Pandas tiene un índice definido explícitamente asociado con los valores.**

El índice explícito le da al objeto Series capacidades adicionales, es decir que el índice no necesita ser un número entero, puede consistir en valores de cualquier tipo.

Crear dándole los valores del índice:

In [8]:
```
data = ['a','b','c','d']
s = pd.Series(data, index=['w','x','y','z'])
s
```

Out[8]:
```
w    a
x    b
y    c
z    d
dtype: object
```

In [9]:
```
data = ['a','b','c','d']
s = pd.Series(data, index=[100,101,102,103])
s
```

Out[9]:
```
100    a
101    b
102    c
103    d
dtype: object
```

Podemos usar índices no contiguos o no secuenciales:

In [10]:
```
s = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
s
```

Out[10]:
```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

Atributos y funciones

index

In [11]:
```
population_dict = {'Buenos Aires': 8332521,
                   'Córdoba': 7448193,
                   'Mendoza': 1965112,
                   'Neuquén': 1955607,
                   'Santa Fé': 1281353}

s = pd.Series(population_dict)
s
```

Out[11]:
```
Buenos Aires    8332521
Córdoba         7448193
Mendoza         1965112
Neuquén         1955607
Santa Fé        1281353
dtype: int64
```

In [12]:
```
print ("Los índices son:\n", s.index)
```

```
Los índices son:
 Index(['Buenos Aires', 'Córdoba', 'Mendoza', 'Neuquén', 'Santa Fé'], dtype='object')
```

values devuelve los datos reales de la serie como una matriz.

In [13]:
```
print ("La serie de datos es:\n", s.values)
```

```
La serie de datos es:
 [8332521 7448193 1965112 1955607 1281353]
```

items

In [14]:
```
s.items
```

Out[14]:
```
<bound method Series.items of Buenos Aires    8332521
Córdoba         7448193
Mendoza         1965112
Neuquén         1955607
Santa Fé        1281353
dtype: int64>
```

axes

In [15]:
```
print ("Los ejes son:\n", s.axes)
```

```
Los ejes son:
 [Index(['Buenos Aires', 'Córdoba', 'Mendoza', 'Neuquén', 'Santa Fé'], dtype='object')]
```

empty devuelve el valor booleano que indica si el objeto está vacío o no. Verdadero indica que el objeto está vacío

In [16]:
```
print ("Está el objeto vacío?\n", s.empty)
```

```
Está el objeto vacío?
 False
```

ndim devuelve el número de dimensiones del objeto. Por definición, una serie es una estructura de datos 1D, por lo que devuelve 1

In [17]:
```
print ("Las dimensiones del objeto:\n", s.ndim)
```

```
Las dimensiones del objeto:
 1
```

size devuelve el tamaño (longitud) de la serie

In [18]:
```
print ("El tamaño del objeto es:\n", s.size)
```

```
El tamaño del objeto es:
 5
```

head() devuelve las primeras n filas (observa los valores de índice).

In [19]:
```
print ("Las primeras dos filas son:\n", s.head(2))
```

```
Las primeras dos filas son:
 Buenos Aires    8332521
Córdoba         7448193
dtype: int64
```

tail() devuelve las últimas n filas (observe los valores de índice). El número predeterminado de elementos para mostrar es 5, pero se puede pasar un número personalizado.

In [20]:
```
print ("Las últimas dos filas son:\n",s.tail(3))
```

```
Las últimas dos filas son:
 Mendoza     1965112
Neuquén     1955607
Santa Fé    1281353
dtype: int64
```

in

In [21]:
```
'Mendoza' in s
```

Out[21]:
```
True
```

keys()

In [22]:
```
s.keys()
```

Out[22]:
```
Index(['Buenos Aires', 'Córdoba', 'Mendoza', 'Neuquén', 'Santa Fé'], dtype='object')
```

concat()

In [23]:
```
s1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
s2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([s1, s2])
```

Out[23]:
```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

Índices duplicados: Si se desea verificar que los índices, en el resultado de pd.concat(), no se superponen, se puede especificar el indicador verify_integrity. Con este atributo en True, la concatenación generará una excepción si hay índices duplicados:

In [24]:
```
s1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
s2 = pd.Series(['D', 'E', 'F'], index=[1, 3, 5])
pd.concat([s1, s2])
```

Out[24]:
```
1    A
2    B
3    C
1    D
3    E
5    F
dtype: object
```

In [25]:
```
try:
    pd.concat([s1, s2], verify_integrity=True)
except ValueError as e:
    print("ValueError:\n", e)
```

```
ValueError:
 Indexes have overlapping values: Int64Index([1, 3], dtype='int64')
```

ValueError: los índices tienen valores superpuestos.

Ignorar el índice: Es posible que el índice no importe y se prefiere ignorar. Puede especificar esta opción usando el atributo ignore_index.

In [26]:
```
s1,s2
```

Out[26]:
```
(1    A
 2    B
 3    C
 dtype: object,
 1    D
 3    E
 5    F
 dtype: object)
```

In [27]:
```
pd.concat([s1, s2], ignore_index=True)
```

Out[27]:
```
0    A
1    B
2    C
3    D
4    E
5    F
dtype: object
```

Con ignore_index establecido en True, la concatenación creará un nuevo índice entero para la Serie resultante.

Acceso a datos de series con posición e índices

Recupera el tercer elemento.

- Si se inserta un **':'** (dos puntos) delante de él, se extraerán todos los elementos de ese índice en adelante.
- Si se utilizan dos parámetros (con **':'** entre ellos), se recuperan los elementos entre los dos índices (sin incluir el índice de detención)

In [28]:
```
s = pd.Series([1,2,3,4,5,6,7,8],index = ['a','b','c','d','e','f','g', 'h'])
s
```

Out[28]:
```
a    1
b    2
c    3
d    4
e    5
f    6
g    7
h    8
dtype: int64
```

In [29]:
```
s[2]
```

Out[29]:
```
3
```

Recupera los primeros tres elementos de la serie:

In [30]:
```
s[:3]
```

Out[30]:
```
a    1
b    2
c    3
dtype: int64
```

Recupera los últimos tres elementos:

In [31]:
```
s[-3:]
```

Out[31]:
```
f    6
g    7
h    8
dtype: int64
```

Recupera desde la posicion 1 hasta la 3. También llamado "Corte por índice entero implícito":

In [32]:
```
s[1:4]
```

Out[32]:
```
b    2
c    3
d    4
dtype: int64
```

Recupera un solo elemento utilizando el valor de la etiqueta de índice:

In [33]:
```
s['a']
```

Out[33]:
```
1
```

Recupera múltiples elementos usando una lista de valores de etiquetas de índice. También llamada "Indexación elegante":

In [34]:
```
s[['a','c','f']]
```

Out[34]:
```
a    1
c    3
f    6
dtype: int64
```

Recupera los valores entre los índices indicados. También llamado "Corte por índice explícito":

In [35]:
```
s['b':'f']
```

Out[35]:
```
b    2
c    3
d    4
e    5
f    6
dtype: int64
```

Recupera los valores que cumplen con la condición. También llamado "Enmascaramiento"

In [36]:
```
s[(s > 3) & (s < 7)]
```

Out[36]:
```
d    4
e    5
f    6
dtype: int64
```

- **Cuando se corta con un índice explícito, s['b':'d'], el índice final se incluye en el segmento.**
- **Cuando se corta con índice implícito, s[1:4], el valor final del índice se excluye del segmento.**

Indexadores: loc e iloc

Atención!: Estas convenciones de segmentación e indexación pueden ser confusas. Por ejemplo:

- si la serie tiene un índice entero explícito, una operación de indexación como s[1] usará los índices **explícitos**,
- mientras que una operación de corte como s[1:3] usará el índice **implícito**.

In [37]:
```
s = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
s
```

Out[37]:
```
1    a
3    b
5    c
dtype: object
```

Índice explícito al indexar:

In [38]:
```
s[1]
```

Out[38]:
```
'a'
```

Índice implícito al cortar:

In [39]:
```
s[1:3]
```

Out[39]:
```
3    b
5    c
dtype: object
```

Debido a esta posible confusión en el caso de los índices de enteros, Pandas proporciona atributos que exponen una interfaz de corte particular a los datos en la Serie.

- El atributo **loc** que permite la indexación y el corte que siempre hace referencia al índice **explícito**:

In [40]:
```
s.loc[1]
```

Out[40]:
```
'a'
```

In [41]:
```
s.loc[1:3]
```

Out[41]:
```
1    a
3    b
dtype: object
```

- El atributo **iloc** permite la indexación y el corte que siempre hace referencia al índice **implícito**:

In [42]:
```
s.iloc[1]
```

Out[42]:
```
'b'
```

In [43]:
```
s.iloc[1:3]
```

Out[43]:
```
3    b
5    c
dtype: object
```

Ufuncs: Conservación de índices

Debido a que Pandas está diseñado para funcionar con NumPy, cualquier ufunc de NumPy funcionará en objetos Pandas Series y DataFrame.

In [44]:
```
import pandas as pd
import numpy as np
```

In [45]:
```
rng = np.random.RandomState(42)
s = pd.Series(rng.randint(0, 10, 4))
s
```

Out[45]:
```
0    6
1    3
2    7
3    4
dtype: int32
```

Si aplicamos un ufunc NumPy sobre cualquiera de estos objetos, el resultado será otro objeto Pandas con los índices conservados:

In [46]:
```
np.exp(s)
```

Out[46]:
```
0     403.428793
1      20.085537
2    1096.633158
3      54.598150
dtype: float64
```

Cualquiera de los ufuncs vistos en Funciones Universales de NumPy se pueden utilizar de manera similar.

Alineación de índices en series

Para operaciones binarias en dos objetos Series o DataFrame, Pandas alineará los índices en el proceso de realizar la operación. Esto es muy conveniente cuando se trabaja con datos incompletos:

In [47]:
```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B
```

Out[47]:
```
0    NaN
1    5.0
2    9.0
3    NaN
dtype: float64
```

In [48]:
```
A/B
```

Out[48]:
```
0    NaN
1    4.0
2    2.0
3    NaN
dtype: float64
```

Cualquier ítem para el cual uno u otro no tiene una entrada se marca con NaN, o "No es un número", que es la forma en que Pandas marca los datos que faltan.

Si usar valores NaN no es el comportamiento deseado, se puede modificar el valor de relleno usando los métodos de objeto apropiados en lugar de los operadores. Por ejemplo, llamar a A.add(B) es equivalente a llamar a A + B, pero permite la especificación explícita opcional del valor de relleno para cualquier elemento en A o B que pueda faltar:

In [49]:
```
A.add(B, fill_value=0)
```

Out[49]:
```
0    2.0
1    5.0
2    9.0
3    5.0
dtype: float64
```

NaN y None

Pandas está diseñado para manejarlos casi indistintamente, convirtiéndolos cuando corresponda:

In [50]:
```
pd.Series([1, np.nan, 2, None])
```

Out[50]:
```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

In [51]:
```
x = pd.Series(range(2), dtype=int)
x
```

Out[51]:
```
0    0
1    1
dtype: int32
```

In [52]:
```
 x[0] = None
```

In [53]:
```
x
```

Out[53]:
```
0    NaN
1    1.0
dtype: float64
```

Pandas convierte automáticamente el valor None en un valor NaN. La siguiente tabla enumera las convenciones de upcasting en Pandas cuando se introducen los valores NaN.

Typeclass	Conversion when storing NAs	NA sentinel value
floating	No change	np.nan
object	No change	None or np.nan
integer	Cast to float64	np.nan
boolean	Cast to object	None or np.nan

Operando con valores nulos

Pandas trata a None y NaN como esencialmente intercambiables para indicar valores faltantes o nulos. Para facilitar esta convención, existen varios métodos útiles para detectar, eliminar y reemplazar valores nulos en las estructuras de datos de Pandas. Ellos son:

isnull(): genera una máscara booleana que indica valores faltantes.

In [54]:
```
datos = pd.Series([1, np.nan, 'hola', None])
datos.isnull()
```

Out[54]:
```
0    False
1     True
2    False
3     True
dtype: bool
```

notnull(): Opuesto a isnull()

In [55]:
```
datos[datos.notnull()]
```

Out[55]:
```
0       1
2    hola
dtype: object
```

Las máscaras booleanas se pueden usar directamente como índice de serie o marco de datos.

dropna(): Devuelve una versión filtrada de los datos.

In [56]:
```
datos.dropna()
```

Out[56]:
```
0       1
2    hola
dtype: object
```

fillna (): devuelve una copia de los datos con los valores faltantes completados o imputados.

In [57]:
```
datos = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
datos
```

Out[57]:
```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

Podemos llenar las entradas de NaN con un solo valor:

In [58]:
```
datos.fillna(4)
```

Out[58]:
```
a    1.0
b    4.0
c    2.0
d    4.0
e    3.0
dtype: float64
```

Podemos especificar un relleno hacia adelante para propagar el valor anterior hacia adelante:

In [59]:
```
datos.fillna(method='ffill')
```

Out[59]:
```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

O podemos especificar un relleno para propagar los valores hacia atrás:

In [60]:
```
datos.fillna(method='bfill')
```

Out[60]:
```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

Operaciones de cadenas vectorizadas

La vectorización de operaciones simplifica la sintaxis de operar en matrices de datos, ya no tenemos que preocuparnos por el tamaño o la forma de la matriz, sino por la operación que queremos que se realice. Para corregir los datos haríamos esto:

In [61]:
```
datos = ['pedro', 'Pablo', 'VILMA', 'gUIDO']
nombres = pd.Series(datos)
nombres
```

Out[61]:
```
0    pedro
1    Pablo
2    VILMA
3    gUIDO
dtype: object
```

In [62]:
```
nombres.str.capitalize()
```

Out[62]:
```
0    Pedro
1    Pablo
2    Vilma
3    Guido
dtype: object
```

In [63]:
```
datos = ['pedro', 'Pablo', None, 'VILMA', 'gUIDO']
datos
```

Out[63]:
```
['pedro', 'Pablo', None, 'VILMA', 'gUIDO']
```

In [64]:
```
nombres = pd.Series(datos)
nombres
```

Out[64]:
```
0    pedro
1    Pablo
2     None
3    VILMA
4    gUIDO
dtype: object
```

In [65]:
```
nombres.str.capitalize()
```

Out[65]:
```
0    Pedro
1    Pablo
2     None
3    Vilma
4    Guido
dtype: object
```

Casi todos los métodos de cadena integrados de Python se reflejan en un método de cadena vectorizado de Pandas:

```
len()  lower()  translate()  islower() ljust() upper() startswith() isupper() rjust() find() endswith() isnumeric() center
() rfind() isalnum() isdecimal() zfill() index() isalpha() split() strip() rindex() isdigit() rsplit() rstrip() capitalize
() isspace() partition() lstrip() swapcase() istitle() rpartition()
```

In [66]:
```
famosos = pd.Series(['Jack Nicholson', 'Dustin Hoffman', 'Tom Hanks','Johnny Depp',
                     'Anthony Hopkins','Richard Gere'])
```

lower()

In [67]:
```
famosos.str.lower()
```

Out[67]:
```
0     jack nicholson
1     dustin hoffman
2          tom hanks
3        johnny depp
4    anthony hopkins
5       richard gere
dtype: object
```

len()

In [68]:
```
famosos.str.len()
```

Out[68]:
```
0    14
1    14
2     9
3    11
4    15
5    12
dtype: int64
```

startswith()

In [69]:
```
famosos.str.startswith('T')
```

Out[69]:
```
0    False
1    False
2     True
3    False
4    False
5    False
dtype: bool
```

split()

```
In [70]: famosos.str.split()
```

```
Out[70]: 0     [Jack, Nicholson]
         1     [Dustin, Hoffman]
         2          [Tom, Hanks]
         3        [Johnny, Depp]
         4    [Anthony, Hopkins]
         5       [Richard, Gere]
         dtype: object
```

Métodos que usan expresiones regulares

Existen varios métodos que aceptan expresiones regulares para examinar el contenido de cada elemento de cadena y siguen algunas de las convenciones de la API del módulo re integrado de Python.

Método	Descripción
match()	Call re.match() on each element, returning a Boolean.
extract()	Call re.match() on each element, returning matched groups as strings.
findall()	Call re.findall() on each element.
replace()	Replace occurrences of pattern with some other string.
contains()	Call re.search() on each element, returning a Boolean.
count()	Count occurrences of pattern.
split()	Equivalent to str.split(), but accepts regexps.
rsplit()	Equivalent to str.rsplit(), but accepts regexps.

extract()

```
In [71]: famosos.str.extract('([A-Za-z]+)')
```

Out[71]:

	0
0	Jack
1	Dustin
2	Tom
3	Johnny
4	Anthony
5	Richard

Extraer el primer nombre de cada uno solicitando un grupo contiguo de caracteres al comienzo de cada elemento.

findall()

```
In [72]: famosos.str.findall(r'^[^AEIOU].*[^aeiou]$')
```

```
Out[72]: 0    [Jack Nicholson]
         1    [Dustin Hoffman]
         2         [Tom Hanks]
         3       [Johnny Depp]
         4                  []
         5                  []
         dtype: object
```

```
In [73]: famosos
```

```
Out[73]: 0     Jack Nicholson
         1     Dustin Hoffman
         2          Tom Hanks
         3        Johnny Depp
         4    Anthony Hopkins
         5       Richard Gere
         dtype: object
```

Encontrar todos los nombres que comienzan o terminan con una consonante, haciendo uso de los caracteres de expresión regular de inicio de cadena (^) y final de cadena ($)

Otros métodos

Hay algunos métodos misceláneos que permiten otras operaciones convenientes.

Method	Description
get()	Index each element
slice()	Slice each element
slice_replace()	Replace slice in each element with passed value
cat()	Concatenate strings
repeat()	Repeat values

Method	Description
normalize()	Return Unicode form of string
pad()	Add whitespace to left, right, or both sides of strings
wrap()	Split long strings into lines with length less than a given width
join()	Join strings in each element of the Series with passed separator
get_dummies()	Extract dummy variables as a DataFrame

Acceso y corte de elementos cadena vectorizados

In [74]:
```
famosos.str[0:3]
```

Out[74]:
```
0    Jac
1    Dus
2    Tom
3    Joh
4    Ant
5    Ric
dtype: object
```

slice()

Recupera los primeros tres caracteres de la matriz.

In [75]:
```
famosos.str.slice(0, 3)
```

Out[75]:
```
0    Jac
1    Dus
2    Tom
3    Joh
4    Ant
5    Ric
dtype: object
```

Las operaciones **get()** y **slice()**, permiten el acceso a elementos vectorizados desde la matriz. También permiten acceder a elementos de arrays devueltos por split():

In [76]:
```
famosos
```

Out[76]:
```
0     Jack Nicholson
1     Dustin Hoffman
2          Tom Hanks
3        Johnny Depp
4    Anthony Hopkins
5       Richard Gere
dtype: object
```

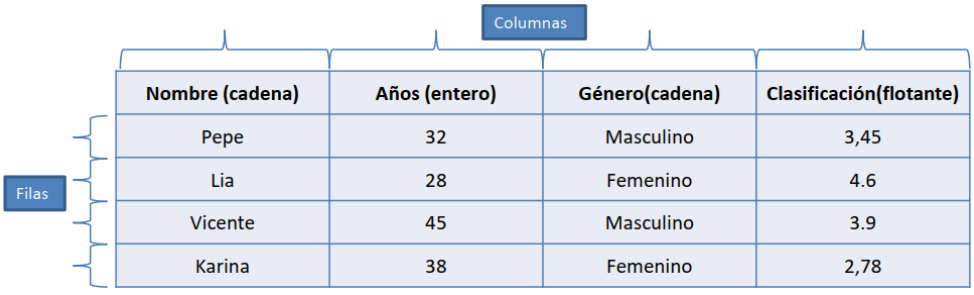
Recuperar el apellido de cada entrada:

In [77]:
```
famosos.str.split().str.get(-1)
```

Out[77]:
```
0    Nicholson
1      Hoffman
2        Hanks
3         Depp
4      Hopkins
5         Gere
dtype: object
```

Dataframe

Data Frame es una matriz bidimensional con datos heterogéneos. Por ejemplo:



Características de DataFrame

- Potencialmente las columnas son de diferentes tipos de datos
- Tamaño: mutable

- Ejes etiquetados (filas y columnas)
- Puede realizar operaciones aritméticas en filas y columnas

- **Se puede crear un DataFrame de pandas utilizando el siguiente constructor: pandas.DataFrame(data, index, columns, dtype, copy)**
- **Se puede crear un DataFrame de pandas usando varias entradas como: list, dict, Series, arrays de Numpy (ndarrays), otro DataFrame**

Crear un dataframe vacío

In [78]:
```
df = pd.DataFrame()
df
```

Out[78]:

Crear un dataframe a partir de listas

In [79]:
```
data = [1,2,3,4,5]
df = pd.DataFrame(data)
df
```

Out[79]:

	0
0	1
1	2
2	3
3	4
4	5

In [80]:
```
type(df)
```

Out[80]: pandas.core.frame.DataFrame

In [81]:
```
data = [['Ale',10],['Pepe',12],['Zule',13]]
df = pd.DataFrame(data,columns=['Nombre','Edad'])
df
```

Out[81]:

	Nombre	Edad
0	Ale	10
1	Pepe	12
2	Zule	13

Crear un dataframe desde un diccionario

- **Todos los arrays deben ser de la misma longitud.**
- **Si se pasa el índice, entonces la longitud del índice debe ser igual a la longitud de las matrices.**
- **Si no se pasa ningún índice, entonces, por defecto, el índice será el rango (n), donde n es la longitud de la matriz.**

In [82]:
```
data = {'Nombre':['Tomy', 'Juan', 'Silvio', 'Ricky'],'Edad':[28,34,29,42]}
df = pd.DataFrame(data)
df
```

Out[82]:

	Nombre	Edad
0	Tomy	28
1	Juan	34
2	Silvio	29
3	Ricky	42

Observa los valores 0,1,2,3. Son el índice predeterminado asignado a cada uno utilizando el rango de funciones (n)

In [83]:
```
data = {'Nombre':['Tomy', 'Juan', 'Silvio', 'Ricky'],
        'Edad':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
df
```

Out[83]:

	Nombre	Edad
rank1	Tomy	28
rank2	Juan	34
rank3	Silvio	29
rank4	Ricky	42

Observa que index asigna una etiqueta a cada fila

Crear un dataframe de una lista de dicccionarios

La lista de diccionarios se puede pasar como datos de entrada para crear un DataFrame. Las claves del diccionario se toman por defecto como nombres de columna.

In [84]:
```
data = [{'a': 1, 'b': 2},
        {'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
df
```

Out[84]:

	a	b	c
0	1	2	NaN
1	5	10	20.0

In [85]:
```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['primera', 'segunda'])
df
```

Out[85]:

	a	b	c
primera	1	2	NaN
segunda	5	10	20.0

Observa que NaN (no es un número) se agrega en las áreas faltantes.

Crear un dataframe pasando una lista de diccionarios y las etiquetas de los índices de fila.

In [86]:
```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df1 = pd.DataFrame(data, index=['primera', 'segunda'], columns=['a', 'b'])
df1
```

Out[86]:

	a	b
primera	1	2
segunda	5	10

In [87]:
```
df2 = pd.DataFrame(data, index=['primera', 'segunda'], columns=['a', 'b1'])
df2
```

Out[87]:

	a	b1
primera	1	NaN
segunda	5	NaN

Observa que mientras el Data Frame df1 se crea con índices de columna iguales a las claves del diccionario, por lo que no se agrega NaN, el Data Frame df2 se crea con un índice de columna que no es clave del diccionario, así se agregaron los NaN.

Crear un dataframe desde un objeto series

In [88]:
```
poblacion_dict = {'Buenos Aires':8332521,'Córdoba':7448193,
                  'Mendoza':1965112,'Neuquén':1955607,'Santa Fé':1281353}
s1 = pd.Series(poblacion_dict)
s1
```

Out[88]:
```
Buenos Aires    8332521
Córdoba         7448193
Mendoza         1965112
Neuquén         1955607
Santa Fé        1281353
dtype: int64
```

In [89]:
```
type(s1)
```

Out[89]:
```
pandas.core.series.Series
```

In [90]:
```
pd.DataFrame(s1, columns=['población'])
```

Out[90]:

	población
Buenos Aires	8332521
Córdoba	7448193
Mendoza	1965112
Neuquén	1955607
Santa Fé	1281353

In [91]:
```
type(pd.DataFrame(s1, columns=['población']))
```

Out[91]:
```
pandas.core.frame.DataFrame
```

Crear un dataframe desde series de diccionarios

La serie de diccionario se puede pasar para formar un dataframe. El índice resultante es la unión de todos los índices de serie pasados.

In [92]:
```
poblacion_dict = {'Buenos Aires':8332521,'Córdoba':7448193,
                  'Mendoza':1965112,'Neuquén':1955607,'Santa Fé':1281353}
s1 = pd.Series(poblacion_dict)
s1
```

Out[92]:
```
Buenos Aires    8332521
Córdoba         7448193
Mendoza         1965112
Neuquén         1955607
Santa Fé        1281353
dtype: int64
```

In [93]:
```
area_dict = {'Buenos Aires':423967, 'Córdoba':695662, 'Mendoza':141297,
             'Neuquén':170312, 'Santa Fé':149995}
s2 = pd.Series(area_dict)
s2
```

Out[93]:
```
Buenos Aires    423967
Córdoba         695662
Mendoza         141297
Neuquén         170312
Santa Fé        149995
dtype: int64
```

In [94]:
```
provincias = pd.DataFrame({'población': s1,'área': s2})
provincias
```

Out[94]:

	población	área
Buenos Aires	8332521	423967
Córdoba	7448193	695662
Mendoza	1965112	141297
Neuquén	1955607	170312
Santa Fé	1281353	149995

In [95]:
```
provincias.index
```

Out[95]:
```
Index(['Buenos Aires', 'Córdoba', 'Mendoza', 'Neuquén', 'Santa Fé'], dtype='object')
```

In [96]:
```
provincias.columns
```

Out[96]:
```
Index(['población', 'área'], dtype='object')
```

In [97]:
```
provincias['área']
```

Out[97]:
```
Buenos Aires    423967
Córdoba         695662
Mendoza         141297
Neuquén         170312
Santa Fé        149995
Name: área, dtype: int64
```

Un dataframe asigna 'serie' a una columna de datos:

In [98]:
```
type(provincias['área'])
```

Out[98]:
```
pandas.core.series.Series
```

Observa: para la serie 'one', no se ha pasado la etiqueta 'd' , pero en el resultado, para la etiqueta d , se agrega un NaN.

In [99]:
```
data = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(data)
df
```

Out[99]:

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	NaN	4

Crear un dataframe con NumPy

Dada una matriz bidimensional de datos, podemos crear un DataFrame con cualquier nombre de columna e índice especificado. Si se omite, se utilizará un índice entero para cada:

In [100]:
```
import numpy as np

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

Out[100]:

	foo	bar
a	0.940852	0.831419
b	0.177027	0.038791
c	0.180185	0.198167

Crear un a partir de una matriz estructurada

In [101]:
```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
pd.DataFrame(A)
```

Out[101]:

	A	B
0	0	0.0
1	0	0.0
2	0	0.0

Al finalizar ésta sección se verán algunos ejemplos de aplicación de Pandas a datos externos.

Atributos y funciones

index()

In [102]:
```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

Out[102]:
```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Tanto los objetos Series como DataFrame contienen un índice explícito que le permite hacer referencia y modificar datos. Este objeto Index se puede considerar como una matriz inmutable o técnicamente, un conjunto múltiple, ya que los objetos Index pueden contener valores repetidos. Por ejemplo, el objeto Index en muchos sentidos funciona como una matriz.

In [103]:
```
ind[1]
```

Out[103]:
```
3
```

In [104]:
```
ind[::2]
```

Out[104]:
```
Int64Index([2, 5, 11], dtype='int64')
```

Una diferencia entre los objetos Index y las matrices NumPy es que los índices son inmutables, es decir, no se pueden modificar por los medios normales:

In [105]:
```
ind[1] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [105], in <cell line: 1>()
----> 1 ind[1] = 0

File C:\anaconda3\lib\site-packages\pandas\core\indexes\base.py:5021, in Index.__setitem__(self, key, value)
   5019 @final
   5020 def __setitem__(self, key, value):
-> 5021     raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

Los objetos de Pandas están diseñados para facilitar operaciones entre conjuntos de datos, que dependen de la aritmética de conjuntos:

In [106]:
```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
indA.intersection(indB)
```

Out[106]:
```
Int64Index([3, 5, 7], dtype='int64')
```

In [107]:
```
indA.union(indB)
```

Out[107]:
```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

In [108]:
```
indA.symmetric_difference(indB)
```

Out[108]:
```
Int64Index([1, 2, 9, 11], dtype='int64')
```

T transpone la matriz de datos

In [109]:
```
d = {'Nombre':pd.Series(['Tomy','Juan','Ricky','Vilma','Silvio','Sara','José']),
     'Edad':pd.Series([25,26,25,23,30,29,23]),
     'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}

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

Out[109]:

	Nombre	Edad	Rating
0	Tomy	25	4.23
1	Juan	26	3.24
2	Ricky	25	3.98
3	Vilma	23	2.56
4	Silvio	30	3.20
5	Sara	29	4.60
6	José	23	3.80

In [110]:
```
df.T
```

Out[110]:

	0	1	2	3	4	5	6
Nombre	Tomy	Juan	Ricky	Vilma	Silvio	Sara	José
Edad	25	26	25	23	30	29	23
Rating	4.23	3.24	3.98	2.56	3.2	4.6	3.8

describe() calcula un resumen de estadísticas pertenecientes a las columnas del DataFrame. Esta función proporciona los valores medios, estándar e IQR . Excluye las columnas de caracteres y el resumen dado es sobre columnas numéricas.

In [111]:
```
df.describe()
```

Out[111]:

	Edad	Rating
count	7.000000	7.000000
mean	25.857143	3.658571
std	2.734262	0.698628
min	23.000000	2.560000
25%	24.000000	3.220000
50%	25.000000	3.800000
75%	27.500000	4.105000
max	30.000000	4.600000

In [112]:
```
df.describe(include=['object'])
```

Out[112]:

	Nombre
count	7
unique	7
top	Tomy
freq	1

In [113]:
```
df.describe(include='all')
```

Out[113]:

	Nombre	Edad	Rating
count	7	7.000000	7.000000
unique	7	NaN	NaN
top	Tomy	NaN	NaN
freq	1	NaN	NaN
mean	NaN	25.857143	3.658571
std	NaN	2.734262	0.698628
min	NaN	23.000000	2.560000
25%	NaN	24.000000	3.220000
50%	NaN	25.000000	3.800000
75%	NaN	27.500000	4.105000
max	NaN	30.000000	4.600000

'include' es el argumento que se utiliza para transmitir la información necesaria sobre qué columnas se debe resumir.

info() brinda información del dataframe

In [114]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Nombre  7 non-null      object 
 1   Edad    7 non-null      int64  
 2   Rating  7 non-null      float64
dtypes: float64(1), int64(1), object(1)
memory usage: 296.0+ bytes
```

axes devuelve la lista de etiquetas de eje de fila y etiquetas de eje de columna.

In [115]:
```
df.axes
```

Out[115]:
```
[RangeIndex(start=0, stop=7, step=1),
 Index(['Nombre', 'Edad', 'Rating'], dtype='object')]
```

dtypes

In [116]:
```
df.dtypes
```

Out[116]:
```
Nombre     object
Edad        int64
Rating    float64
dtype: object
```

empty

In [117]:
```
df.empty
```

Out[117]:
```
False
```

ndim

In [118]:
```
df.ndim
```

Out[118]:
```
2
```

shape devuelve una tupla que representa la dimensión del Data Frame. Tupla (a, b), donde a representa el número de filas y b representa el número de columnas

In [119]:
```
df.shape
```

Out[119]:
```
(7, 3)
```

size devuelve la cantidad de elementos del DataFrame.

In [120]:
```
df.size
```

Out[120]:
```
21
```

values devuelve los datos del DataFrame como un ndarray.

In [121]:
```
df.values
```

Out[121]:
```
array([['Tomy', 25, 4.23],
       ['Juan', 26, 3.24],
       ['Ricky', 25, 3.98],
       ['Vilma', 23, 2.56],
       ['Silvio', 30, 3.2],
       ['Sara', 29, 4.6],
       ['José', 23, 3.8]], dtype=object)
```

In [122]:
```
df.values[3]
```

Out[122]:
```
array(['Vilma', 23, 2.56], dtype=object)
```

head() devuelve las primeras n filas (observa los valores de índice). El número predeterminado de elementos para mostrar es 5, pero puedes pasar un número personalizado.

In [123]:
```
df.head(2)
```

Out[123]:

	Nombre	Edad	Rating
0	Tomy	25	4.23
1	Juan	26	3.24

tail() devuelve las últimas n filas.

In [124]:
```
df.tail(3)
```

Out[124]:

	Nombre	Edad	Rating
4	Silvio	30	3.2
5	Sara	29	4.6
6	José	23	3.8

Agregar columnas

In [125]:
```
d = {'uno' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'dos' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
df
```

Out[125]:

	uno	dos
a	1.0	1
b	2.0	2
c	3.0	3
d	NaN	4

Agrega una nueva columna pasada como serie:

In [126]:
```
df['tres']=pd.Series([10,20,30],index=['a','b','c'])
df
```

Out[126]:

	uno	dos	tres
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

Agrega una nueva columna usando las columnas existentes en el dataframe:

In [127]:
```
df['cuatro']=df['uno'] + df['tres']
df
```

Out[127]:

	uno	dos	tres	cuatro
a	1.0	1	10.0	11.0
b	2.0	2	20.0	22.0
c	3.0	3	30.0	33.0
d	NaN	4	NaN	NaN

Eliminar columnas

In [128]:
```
d = {'uno' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'dos' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
     'tres' : pd.Series([10,20,30], index=['a','b','c'])}
df = pd.DataFrame(d)
df
```

Out[128]:

	uno	dos	tres
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

del:

Elimina una columna y no devuelve la columna eliminada:

In [129]:
```
del df['uno']
df
```

Out[129]:

	dos	tres
a	1	10.0
b	2	20.0
c	3	30.0
d	4	NaN

pop()

Elimina una columna y devuelve la columna eliminada:

In [130]:
```
df.pop('dos')
df
```

Out[130]:

	tres
a	10.0
b	20.0
c	30.0
d	NaN

Agregar filas

append() agregará las filas al final.

In [131]:
```
import warnings
warnings.filterwarnings('ignore')

df1 = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])
df = df1.append(df2)
df
```

Out[131]:

	a	b
0	1	2
1	3	4
0	5	6
1	7	8

Eliminar filas

drop() si la etiqueta está duplicada, se eliminarán varias filas.

In [132]:
```
df = df.drop(0)
df
```

Out[132]:

	a	b
1	3	4
1	7	8

Observa: Se eliminaron 2 filas porque esas dos contienen la misma etiqueta 0

Acceso a datos de dataframe con posición e índices

La indexación se refiere a las columnas, el corte (slicing) se refiere a las filas:

In [133]:
```
poblacion_dict = {'Buenos Aires':8332521,'Córdoba':7448193,'Mendoza':1965112,'Neuquén':1955607,'Santa Fé':1281353}
s1 = pd.Series(poblacion_dict)
area_dict = {'Buenos Aires':423967, 'Córdoba':695662, 'Mendoza':141297, 'Neuquén':170312, 'Santa Fé':149995}
s2 = pd.Series(area_dict)
provincias = pd.DataFrame({'población': s1,'área': s2})
provincias
```

Out[133]:

	población	área
Buenos Aires	8332521	423967
Córdoba	7448193	695662
Mendoza	1965112	141297
Neuquén	1955607	170312
Santa Fé	1281353	149995

Pasar un solo "índice" a un DataFrame accede a una columna:

In [134]:
```
provincias['área']
```

Out[134]:
```
Buenos Aires    423967
Córdoba         695662
Mendoza         141297
Neuquén         170312
Santa Fé        149995
Name: área, dtype: int64
```

Los segmentos también pueden referirse a filas por número en lugar de por índice:

In [135]:
```
provincias['Córdoba':'Neuquén']
```

Out[135]:

	población	área
Córdoba	7448193	695662
Mendoza	1965112	141297
Neuquén	1955607	170312

In [136]:
```
provincias[1:3]
```

Out[136]:

	población	área
Córdoba	7448193	695662
Mendoza	1965112	141297

Las operaciones de enmascaramiento directo también se interpretan por filas en lugar de por columnas:

In [137]:
```
provincias[provincias['área'] > 300000]
```

Out[137]:

	población	área
Buenos Aires	8332521	423967
Córdoba	7448193	695662

Al igual que con los objetos Series esta sintaxis también se puede usar para modificar el objeto, en este caso para agregar una nueva columna:

In [138]:
```
provincias['densidad'] = provincias['población'] / provincias['área']
provincias
```

Out[138]:

	población	área	densidad
Buenos Aires	8332521	423967	19.653702
Córdoba	7448193	695662	10.706626
Mendoza	1965112	141297	13.907670
Neuquén	1955607	170312	11.482497
Santa Fé	1281353	149995	8.542638

In [139]:
```
provincias.values
```

Out[139]:
```
array([[8.33252100e+06, 4.23967000e+05, 1.96537018e+01],
       [7.44819300e+06, 6.95662000e+05, 1.07066262e+01],
       [1.96511200e+06, 1.41297000e+05, 1.39076697e+01],
       [1.95560700e+06, 1.70312000e+05, 1.14824968e+01],
       [1.28135300e+06, 1.49995000e+05, 8.54263809e+00]])
```

In [140]:
```
provincias.values[2]
```

Out[140]:
```
array([1.96511200e+06, 1.41297000e+05, 1.39076697e+01])
```

loc selección pasando la etiqueta de fila a una función loc

In [141]:
```
datos = {'uno':pd.Series([1, 2, 3],index=['a', 'b', 'c']),'dos':pd.Series([1, 2, 3, 4],index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(datos)
df
```

Out[141]:

	uno	dos
a	1.0	1
b	2.0	2
c	3.0	3
d	NaN	4

In [142]:
```
df.loc['b']
```

Out[142]:
```
uno    2.0
dos    2.0
Name: b, dtype: float64
```

El resultado es una serie con etiquetas con el nombres de columna del DataFrame. Y el nombre de la serie es la etiqueta con la que se recupera:

In [143]:
```
type(df.loc['b'])
```

Out[143]:
```
pandas.core.series.Series
```

Recuperamos los datos con el atributo de nombres de columna que son cadenas:

In [144]:
```
df.uno
```

Out[144]:
```
a    1.0
b    2.0
c    3.0
d    NaN
Name: uno, dtype: float64
```

In [145]:
```
type(df.uno)
```

Out[145]:
```
pandas.core.series.Series
```

In [146]:
```
poblacion_dict = {'Buenos Aires':8332521,'Córdoba':7448193,'Mendoza':1965112,'Neuquén':1955607,'Santa Fé':1281353}
s1 = pd.Series(poblacion_dict)
area_dict = {'Buenos Aires':423967, 'Córdoba':695662, 'Mendoza':141297, 'Neuquén':170312, 'Santa Fé':149995}
s2 = pd.Series(area_dict)
provincias = pd.DataFrame({'población': s1,'área': s2})
provincias
```

Out[146]:

	población	área
Buenos Aires	8332521	423967
Córdoba	7448193	695662
Mendoza	1965112	141297
Neuquén	1955607	170312
Santa Fé	1281353	149995

Pasar un solo índice accede a una fila:

In [147]:
```
provincias.loc['Mendoza']
```

Out[147]:
```
población    1965112
área          141297
Name: Mendoza, dtype: int64
```

Con el indexador loc podemos combinar el enmascaramiento y la indexación elegante:

In [148]:
```
provincias.loc[:'Mendoza', :'población']
```

Out[148]:

	población
Buenos Aires	8332521
Córdoba	7448193
Mendoza	1965112

In [149]:
```
provincias.loc[provincias['población'] > 3000000, ['población', 'área']]
```

Out[149]:

	población	área
Buenos Aires	8332521	423967
Córdoba	7448193	695662

iloc Selección de fila pasando la ubicación entera a una función iloc:

In [150]:
```
d = {'uno' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'dos' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
    'tres' : pd.Series([4, 5, 6, 7], index=['a', 'b', 'c', 'd']),
    'cuatro' : pd.Series([5, 8, 7], index=['a', 'c', 'd'])}
df = pd.DataFrame(d)
df
```

Out[150]:

	uno	dos	tres	cuatro
a	1.0	1	4	5.0
b	2.0	2	5	NaN
c	3.0	3	6	8.0
d	NaN	4	7	7.0

In [151]:
```
df.iloc[2]
```

Out[151]:
```
uno       3.0
dos       3.0
tres      6.0
cuatro    8.0
Name: c, dtype: float64
```

*Se pueden seleccionar varias filas con el operador **':'***

In [152]:
```
df.iloc[2:4]
```

Out[152]:

	uno	dos	tres	cuatro
c	3.0	3	6	8.0
d	NaN	4	7	7.0

Formato fila columna. Se pueden seleccionar modificar valores:

In [153]:
```
df.iloc[1,2] = 90
df
```

Out[153]:

	uno	dos	tres	cuatro
a	1.0	1	4	5.0
b	2.0	2	90	NaN
c	3.0	3	6	8.0
d	NaN	4	7	7.0

In [154]:
```
poblacion_dict = {'Buenos Aires':8332521,'Córdoba':7448193,'Mendoza':1965112,'Neuquén':1955607,'Santa Fé':1281353}
s1 = pd.Series(poblacion_dict)
area_dict = {'Buenos Aires':423967, 'Córdoba':695662, 'Mendoza':141297, 'Neuquén':170312, 'Santa Fé':149995}
s2 = pd.Series(area_dict)
provincias = pd.DataFrame({'población': s1,'área': s2})
provincias
```

Out[154]:

	población	área
Buenos Aires	8332521	423967
Córdoba	7448193	695662
Mendoza	1965112	141297
Neuquén	1955607	170312
Santa Fé	1281353	149995

In [155]:
```
provincias.iloc[:3, :2]
```

Out[155]:

	población	área
Buenos Aires	8332521	423967
Córdoba	7448193	695662
Mendoza	1965112	141297

Ufuncs: Conservación de índices

Debido a que Pandas está diseñado para funcionar con NumPy, cualquier ufunc de NumPy funcionará en objetos Pandas Series y DataFrame.

In [156]:
```
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'])
df
```

Out[156]:

	A	B	C	D
0	6	9	2	6
1	7	4	3	7
2	7	2	5	4

In [157]:
```
np.sin(df * np.pi / 4)
```

Out[157]:

	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

In [158]:
```
type(np.sin(df * np.pi / 4))
```

Out[158]: `pandas.core.frame.DataFrame`

UFuncs: Alineación de índices

In [159]:
```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),columns=list('AB'))
A
```

Out[159]:

	A	B
0	1	11
1	5	1

In [160]:
```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),columns=list('BAC'))
B
```

Out[160]:

	B	A	C
0	4	0	9
1	5	8	0
2	9	2	6

Cuando realiza operaciones en dataframes, se produce un tipo similar de alineación tanto para las columnas como para los índices. Obsérvese que los índices están alineados correctamente independientemente del orden en los dos objetos:

In [161]:
```
A + B
```

Out[161]:

	A	B	C
0	1.0	15.0	NaN
1	13.0	6.0	NaN
2	NaN	NaN	NaN

Completamos con la media de todos los valores en A (que calculamos apilando primero las filas de A). Como en Series, se puede utilizar el método aritmético del objeto asociado y pasar cualquier valor de relleno para las entradas faltantes:

In [162]:
```
fill = A.stack().mean()
fill
```

Out[162]: `4.5`

In [163]:
```
A.add(B, fill_value=fill)
```

Out[163]:

	A	B	C
0	1.0	15.0	13.5
1	13.0	6.0	4.5
2	6.5	13.5	10.5

Mapeo entre operadores de Python y métodos de Pandas

Operadores de Python	Métodos de Pandas

Operadores de Python	Métodos de Pandas
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Ufuncs: Operaciones entre DataFrame y Series

Cuando realiza operaciones entre un dataframe y una serie, la alineación del índice y la columna se mantiene de manera similar. Las operaciones entre estos objetos son similares a las operaciones entre un bidimensional y un unidimensional.

In [164]:
```
A = rng.randint(10, size=(3, 4))
A
```

Out[164]:
```
array([[3, 8, 2, 4],
       [2, 6, 4, 8],
       [6, 1, 3, 8]])
```

De acuerdo con las reglas de transmisión, la resta entre una matriz bidimensional y una de sus filas se aplica por filas:

In [165]:
```
A - A[0]
```

Out[165]:
```
array([[ 0,  0,  0,  0],
       [-1, -2,  2,  4],
       [ 3, -7,  1,  4]])
```

Si se desea operar por columnas, pueden usarse los métodos de objeto mencionados mientras especifica el atributo del eje:

In [166]:
```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]
```

Out[166]:

	Q	R	S	T
0	0	0	0	0
1	-1	-2	2	4
2	3	-7	1	4

In [167]:
```
df.subtract(df['R'], axis=0)
```

Out[167]:

	Q	R	S	T
0	-5	0	-6	-4
1	-4	0	-2	2
2	5	0	2	7

In [168]:
```
x = df.iloc[0, ::2]
x
```

Out[168]:
```
Q    3
S    2
Name: 0, dtype: int32
```

Estas operaciones entre dataFrame y series, alinearán automáticamente los índices entre los dos elementos:

In [169]:
```
df - x
```

Out[169]:

	Q	R	S	T
0	0.0	NaN	0.0	NaN
1	-1.0	NaN	2.0	NaN
2	3.0	NaN	1.0	NaN

NaN y None

In [170]:
```
df = pd.DataFrame([[1, np.nan, 2], [2, 3, 5], [np.nan, 4, 6]])
df
```

Out[170]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

dropna()

No se pueden eliminar valores individuales de un dataFrame; solo filas o columnas completas:

In [171]:
```
df.dropna()
```

Out[171]:

	0	1	2
1	2.0	3.0	5

In [172]:
```
df.dropna(axis='columns')
```

Out[172]:

	2
0	2
1	5
2	6

Se puede eliminar los valores NaN a lo largo de un eje; axis=1 elimina todas las columnas que contienen un valor nulo:

In [173]:
```
df.dropna(axis=1)
```

Out[173]:

	2
0	2
1	5
2	6

how

how='all' solo eliminará filas/columnas que sean todos valores nulos. El valor predeterminado es how='any':

In [174]:
```
df[3] = np.nan
df
```

Out[174]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [175]:
```
df.dropna(axis='columns', how='all')
```

Out[175]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

thresh

'tresh' permite especificar un número mínimo de valores no nulos para que se conserve la fila/columna:

In [176]:
```
df
```

Out[176]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [177]:
```
df.dropna(axis='rows', thresh=3)
```

Out[177]:

	0	1	2	3
1	2.0	3.0	5	NaN

fillna()

Si un valor anterior no está disponible durante un llenado hacia adelante, el valor NaN permanece:

In [178]:
```
df.fillna(method='ffill', axis=1)
```

Out[178]:

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

concat()

In [179]:
```
def make_df(cols, ind):
    data = {c: [str(c) + str(i) for i in ind]
        for c in cols}
    return pd.DataFrame(data, ind)
```

In [180]:
```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
df1,df2
```

Out[180]:
```
(    A   B
 1  A1  B1
 2  A2  B2,
     A   B
 3  A3  B3
 4  A4  B4)
```

La concatenación se realiza por filas dentro del marco de datos (es decir, eje = 0):

In [181]:
```
pd.concat([df1, df2])
```

Out[181]:

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

pd.concat permite la especificación de un eje a lo largo del cual tendrá lugar la concatenación:

In [182]:
```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
pd.concat([df3, df4], axis='columns')
```

Out[182]:

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

In [183]:
```
df1 = make_df('ABC', [1, 2])
df2 = make_df('BCD', [3, 4])
df=pd.concat([df1, df2])
df
```

Out[183]:

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

De forma predeterminada, las entradas para las que no hay datos disponibles se rellenan con valores NaN. Para cambiar esto, podemos especificar una de varias opciones para los parámetros _join y **join_axes** de la función de concatenación. Por defecto, la unión es una unión de las columnas de entrada **(join='outer')**, pero podemos cambiar esto a una intersección de las columnas usando **join='inner'**

In [184]:
```
df = pd.concat([df1, df2], join='inner')
df
```

Out[184]:

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

Combinación de conjuntos de datos

merge()

In [185]:
```
df1 = pd.DataFrame({'empleado': ['Bob', 'Juan', 'Lisa', 'Susana'],
                    'grupo':['Contabilidad','Ingeniería','Ingeniería','RRHH']})
df2 = pd.DataFrame({'empleado': ['Lisa', 'Bob', 'Juan', 'Susana'],
                    'fecha_de_contrato': [2004, 2008, 2012, 2014]})
df1
```

Out[185]:

	empleado	grupo
0	Bob	Contabilidad
1	Juan	Ingeniería
2	Lisa	Ingeniería
3	Susana	RRHH

In [186]:
```
df2
```

Out[186]:

	empleado	fecha_de_contrato
0	Lisa	2004
1	Bob	2008
2	Juan	2012
3	Susana	2014

Unión uno a uno: merge() reconoce que cada dataFrame tiene una columna de "empleado" y se une nediante esta columna como clave. El resultado de la fusión es un nuevo DataFrame que combina la información de las dos entradas. Tenga en cuenta que el orden de las entradas en cada columna no se mantiene necesariamente. La fusión en general descarta el índice, excepto en el caso especial de las fusiones por índice.

In [187]:
```
df = pd.merge(df1, df2)
df
```

Out[187]:

	empleado	grupo	fecha_de_contrato
0	Bob	Contabilidad	2008
1	Juan	Ingeniería	2012
2	Lisa	Ingeniería	2004
3	Susana	RRHH	2014

Unión de muchos a uno:

In [188]:
```
df3 = pd.DataFrame({'grupo': ['Contabilidad', 'Ingeniería', 'RRHH'],'supervisor': ['Carly', 'Guido', 'Esteban']})
df3
```

Out[188]:

	grupo	supervisor
0	Contabilidad	Carly
1	Ingeniería	Guido
2	RRHH	Esteban

In [189]:
```
pd.merge(df, df3)
```

Out[189]:

	empleado	grupo	fecha_de_contrato	supervisor
0	Bob	Contabilidad	2008	Carly
1	Juan	Ingeniería	2012	Guido
2	Lisa	Ingeniería	2004	Guido
3	Susana	RRHH	2014	Esteban

Unión de muchos a muchos: Si la columna clave en la matriz izquierda y derecha contiene duplicados, entonces el resultado es una combinación de muchos a muchos. En el ejemplo, al realizar una unión de muchos a muchos, podemos recuperar las habilidades asociadas con cualquier persona individual.

In [190]:
```
df4 = pd.DataFrame({'grupo': ['Contabilidad', 'Contabilidad', 'Ingeniería', 'Ingeniería', 'RRHH', 'RRHH'],
                    'habilidades': ['matemáticas', 'hojas de cálculo', 'codificación', 'linux','hojas de cálculo',
                                    'organización']})
df4
```

Out[190]:

	grupo	habilidades
0	Contabilidad	matemáticas
1	Contabilidad	hojas de cálculo
2	Ingeniería	codificación
3	Ingeniería	linux
4	RRHH	hojas de cálculo
5	RRHH	organización

In [191]:
```
pd.merge(df1, df4)
```

Out[191]:

	empleado	grupo	habilidades
0	Bob	Contabilidad	matemáticas
1	Bob	Contabilidad	hojas de cálculo
2	Juan	Ingeniería	codificación
3	Juan	Ingeniería	linux
4	Lisa	Ingeniería	codificación
5	Lisa	Ingeniería	linux
6	Susana	RRHH	hojas de cálculo
7	Susana	RRHH	organización

La palabra clave 'on' especifica explícitamente el nombre de la columna por la cual se desea unir:

In [192]:
```
pd.merge(df1, df2, on='empleado')
```

Out[192]:

	empleado	grupo	fecha_de_contrato
0	Bob	Contabilidad	2008
1	Juan	Ingeniería	2012
2	Lisa	Ingeniería	2004
3	Susana	RRHH	2014

Las palabras clave 'left_on' y 'right_on', funcionan si los dataFrames izquierdo y derecho tienen el nombre de columna especificado:

In [193]:
```
df5 = pd.DataFrame({'nombre': ['Bob', 'Juan', 'Lisa', 'Susana'],'salario': [70000, 80000, 120000, 90000]})
df5
```

Out[193]:

	nombre	salario
0	Bob	70000
1	Juan	80000
2	Lisa	120000
3	Susana	90000

In [194]:
```
pd.merge(df1, df5, left_on="empleado", right_on="nombre")
```

Out[194]:

	empleado	grupo	nombre	salario
0	Bob	Contabilidad	Bob	70000
1	Juan	Ingeniería	Juan	80000
2	Lisa	Ingeniería	Lisa	120000
3	Susana	RRHH	Susana	90000

Como el resultado tiene una columna redundante puede quitarse usando el método 'drop()' de dataFrames:

In [195]:
```
pd.merge(df1, df5, left_on="empleado", right_on="nombre").drop('nombre', axis=1)
```

Out[195]:

	empleado	grupo	salario
0	Bob	Contabilidad	70000
1	Juan	Ingeniería	80000
2	Lisa	Ingeniería	120000
3	Susana	RRHH	90000

Fusión por índice:

In [196]:
```
df1a = df1.set_index('empleado')
df2a = df2.set_index('empleado')
pd.merge(df1a, df2a, left_index=True, right_index=True)
```

Out[196]:

	grupo	fecha_de_contrato
empleado		
Bob	Contabilidad	2008
Juan	Ingeniería	2012
Lisa	Ingeniería	2004
Susana	RRHH	2014

El método join() realiza una combinación que une los índices:

In [197]:
```
df1a.join(df2a)
```

Out[197]:

empleado	grupo	fecha_de_contrato
Bob	Contabilidad	2008
Juan	Ingeniería	2012
Lisa	Ingeniería	2004
Susana	RRHH	2014

Puede combinarse left_index con right_on o left_on con right_index para obtener el comportamiento deseado:

In [198]:
```
pd.merge(df1a, df5, left_index=True, right_on='nombre')
```

Out[198]:

	grupo	nombre	salario
0	Contabilidad	Bob	70000
1	Ingeniería	Juan	80000
2	Ingeniería	Lisa	120000
3	RRHH	Susana	90000

In [199]:
```
df1 = pd.DataFrame({'nombre': ['Pedro', 'Pablo', 'Mary'],
                    'comida': ['pescado', 'fruta', 'pan']}, columns=['nombre', 'comida'])
df2 = pd.DataFrame({'nombre': ['Mary', 'Jose'],
                    'bebida': ['agua', 'gaseosa']}, columns=['nombre', 'bebida'])
df1
```

Out[199]:

	nombre	comida
0	Pedro	pescado
1	Pablo	fruta
2	Mary	pan

In [200]:
```
df2
```

Out[200]:

	nombre	bebida
0	Mary	agua
1	Jose	gaseosa

In [201]:
```
pd.merge(df1, df2)
```

Out[201]:

	nombre	comida	bebida
0	Mary	pan	agua

*El resultado contiene la intersección de los dos conjuntos de entradas; esto es lo que se conoce como **'unión interna'**. Puede especificarse explícitamente usando la palabra clave 'how'.*

In [202]:
```
pd.merge(df1, df2, how='inner')
```

Out[202]:

	nombre	comida	bebida
0	Mary	pan	agua

Una combinación 'outer' devuelve una combinación sobre la unión de las columnas de entrada y completa todos los valores faltantes con NaN:

In [203]:
```
pd.merge(df1, df2, how='outer')
```

Out[203]:

	nombre	comida	bebida
0	Pedro	pescado	NaN
1	Pablo	fruta	NaN
2	Mary	pan	agua
3	Jose	NaN	gaseosa

La combinación 'left' y la combinación 'right' devuelven la combinación sobre las entradas izquierda y derecha, respectivamente:

In [204]:
```
pd.merge(df1, df2, how='left')
```

Out[204]:

	nombre	comida	bebida
0	Pedro	pescado	NaN
1	Pablo	fruta	NaN
2	Mary	pan	agua

In [205]:
```
pd.merge(df1, df2, how='right')
```

Out[205]:

	nombre	comida	bebida
0	Mary	pan	agua
1	Jose	NaN	gaseosa

Agregación simple

Agregación	Descripción
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

In [206]:
```
d = {'Nombre':pd.Series(['Tomy','Juan','Ricky','Vilma','Silvio','Sara','José']),
     'Edad':pd.Series([25,26,25,23,30,29,23]),
     'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}
df = pd.DataFrame(d)
df
```

Out[206]:

	Nombre	Edad	Rating
0	Tomy	25	4.23
1	Juan	26	3.24
2	Ricky	25	3.98
3	Vilma	23	2.56
4	Silvio	30	3.20
5	Sara	29	4.60
6	José	23	3.80

sum:

In [207]:
```
df.sum()
```

Out[207]:
```
Nombre    TomyJuanRickyVilmaSilvioSaraJosé
Edad                                   181
Rating                               25.61
dtype: object
```

Suma del eje 1

In [208]:
```
df.sum(1)
```

Out[208]:
```
0    29.23
1    29.24
2    28.98
3    25.56
4    33.20
5    33.60
6    26.80
dtype: float64
```

Observa: sumó los valores numéricos por fila.

mean:

In [209]:
```
df.mean(numeric_only=True)
```

Out[209]:
```
Edad      25.857143
Rating     3.658571
dtype: float64
```

std

In [210]:
```
df.std(numeric_only=True)
```

Out[210]:
```
Edad      2.734262
Rating    0.698628
dtype: float64
```

Agrupación

> • El paso de división implica dividir y agrupar un dataFrame según el valor de la clave especificada.
> • El paso de aplicación implica el cálculo de alguna función, generalmente un agregado, transformación o filtrado, dentro de los grupos individuales.
> • El paso de combinación fusiona los resultados de estas operaciones en una matriz de salida.

groupby()

In [211]:
```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'], 'data': range(6)}, columns=['key', 'data'])
df
```

Out[211]:

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

Se devuelve un objeto DataFrameGroupBy. No realiza ningún cálculo real hasta que se aplica la agregación:

In [212]:
```
df.groupby('key')
```

Out[212]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001C968A578B0>
```

In [213]:
```
df.groupby('key').sum()
```

Out[213]:

	data
key	
A	3
B	5
C	7

aggregate()

In [214]:
```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6), 'data2': rng.randint(0, 10, 6)},
                  columns = ['key', 'data1', 'data2'])

df
```

Out[214]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

El método de agregación() permite calcular todos los agregados a la vez:

In [215]:
```
df.groupby('key').aggregate(['min', np.median, max])
```

Out[215]:

	data1			data2		
	min	**median**	**max**	**min**	**median**	**max**
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Otra forma es pasar los nombres de las columnas de asignación es en un diccionario con las operaciones que se aplicarán en esas columnas:

In [216]:
```
df.groupby('key').aggregate({'data1': 'min', 'data2': 'max'})
```

Out[216]:

key	data1	data2
A	0	5
B	1	7
C	2	9

filter()

Una operación de filtrado permite colocar datos en función de las propiedades del grupo. En este ejemplo queremos mantener todos los grupos en los que la desviación estándar es mayor que a un valor crítico. La función filter() debe devolver un valor booleano que especifique si el grupo pasa el filtrado. Aquí, debido a que el grupo A no tiene una desviación estándar superior a 4, se elimina del resultado.

In [217]:
```
df
```

Out[217]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

In [218]:
```
def filter_func(x):
    return x['data2'].std() > 4

df.groupby('key').std()
```

Out[218]:

key	data1	data2
A	2.12132	1.414214
B	2.12132	4.949747
C	2.12132	4.242641

In [219]:
```
df.groupby('key').filter(filter_func)
```

Out[219]:

	key	data1	data2
1	B	1	0
2	C	2	3
4	B	4	7
5	C	5	9

transform()

In [220]:
```
df
```

Out[220]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Mientras que la agregación debe devolver una versión reducida de los datos, la transformación puede devolver alguna versión transformada de los datos completos para recombinarlos. Para tal transformación, la salida tiene la misma forma que la entrada. Un ejemplo común es centrar los datos restando la media del grupo:

In [221]:
```
df.groupby('key').transform(lambda x: x - x.mean())
```

Out[221]:

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

apply()

apply() permite aplicar una función arbitraria a los resultados del grupo. La función debe tomar un dataFrame y devolver un objeto como un dataFrame, series o un escalar; la operación de combinación se adaptará al tipo de salida devuelta. En el ejemplo, se normaliza la primera columna por la suma de los segundos.

In [222]:
```
def norm_by_data2(x):
    x['data1'] /= x['data2'].sum()
    return x

df.groupby('key').apply(norm_by_data2)
```

Out[222]:

	key	data1	data2
0	A	0.000000	5
1	B	0.142857	0
2	C	0.166667	3
3	A	0.375000	3
4	B	0.571429	7
5	C	0.416667	9

Fechas y horarios en Pandas

In [223]:
```
date = pd.to_datetime("4th of May, 2022")
date
```

Out[223]:
```
Timestamp('2022-05-04 00:00:00')
```

In [224]:
```
date.strftime('%A')
```

Out[224]:
```
'Wednesday'
```

Pueden hacerse operaciones vectorizadas directamente en este mismo objeto:

In [225]:
```
date + pd.to_timedelta(np.arange(12), 'D')
```

Out[225]:
```
DatetimeIndex(['2022-05-04', '2022-05-05', '2022-05-06', '2022-05-07',
               '2022-05-08', '2022-05-09', '2022-05-10', '2022-05-11',
               '2022-05-12', '2022-05-13', '2022-05-14', '2022-05-15'],
              dtype='datetime64[ns]', freq=None)
```

Puede crearse un objeto serie que tenga datos indexados en el tiempo:

In [226]:
```
index = pd.DatetimeIndex(['2022-07-04', '2022-08-04', '2021-07-04', '2021-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
data
```

Out[226]:
```
2022-07-04    0
2022-08-04    1
2021-07-04    2
2021-08-04    3
dtype: int64
```

In [227]:
```
data['2021-08-04': '2022-07-04']
```

Out[227]:
```
2022-07-04    0
2021-08-04    3
dtype: int64
```

In [228]:
```
data['2021']
```

Out[228]:
```
2021-07-04    2
2021-08-04    3
dtype: int64
```

In [229]:
```
pd.date_range('2022-07-03', '2022-07-10')
```

Out[229]:
```
DatetimeIndex(['2022-07-03', '2022-07-04', '2022-07-05', '2022-07-06',
               '2022-07-07', '2022-07-08', '2022-07-09', '2022-07-10'],
              dtype='datetime64[ns]', freq='D')
```

😃 Acceso a datos con Pandas y gráficos con Matplotlib

In [230]:
```
import matplotlib.pyplot as plt
import pandas as pd
```

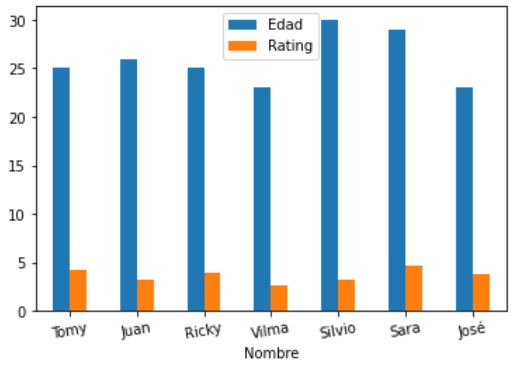
In [231]:
```
data = {'Nombre':pd.Series(['Tomy','Juan','Ricky','Vilma','Silvio','Sara','José']),
        'Edad':pd.Series([25,26,25,23,30,29,23]),
        'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}

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

Out[231]:

	Nombre	Edad	Rating
0	Tomy	25	4.23
1	Juan	26	3.24
2	Ricky	25	3.98
3	Vilma	23	2.56
4	Silvio	30	3.20
5	Sara	29	4.60
6	José	23	3.80

In [232]:
```
df.plot.bar('Nombre', rot=10)
```

Out[232]:
```
<AxesSubplot:xlabel='Nombre'>
```



read_html() permite acceder a los datos de un sitio web.

In [233]:
```
url='https://www.tiobe.com/tiobe-index/'
web=pd.read_html(url, header=0)[0]
df = pd.DataFrame(web)
df.head()
```

Out[233]:

	Sep 2022	Sep 2021	Change	Programming Language	Programming Language.1	Ratings	Change.1
0	1	2	NaN	NaN	Python	15.74%	+4.07%
1	2	1	NaN	NaN	C	13.96%	+2.13%
2	3	3	NaN	NaN	Java	11.72%	+0.60%
3	4	4	NaN	NaN	C++	9.76%	+2.63%
4	5	5	NaN	NaN	C#	4.88%	-0.89%

In [234]:
```
df = df.drop(['Sep 2022', 'Sep 2021', 'Change','Programming Language', 'Change.1'], axis=1)
df.head()
```

Out[234]:

	Programming Language.1	Ratings
0	Python	15.74%
1	C	13.96%
2	Java	11.72%
3	C++	9.76%
4	C#	4.88%

rstrip(): Quitamos el caracter de la derecha y **astype()** permite convertir el valor (texto) en número:

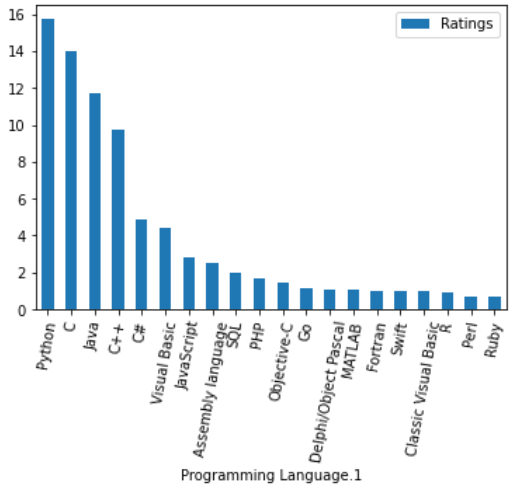
In [235]:
```
df["Ratings"] = df["Ratings"].str.rstrip('%').astype('float')
df.set_index('Programming Language.1',inplace=True)
df.head()
```

Out[235]:

Programming Language.1	Ratings
Python	15.74
C	13.96
Java	11.72
C++	9.76
C#	4.88

In [236]:
```
df.plot.bar(rot=80)
```

Out[236]:
```
<AxesSubplot:xlabel='Programming Language.1'>
```



In [237]:
```
web=pd.read_html(url, header=0)[2]
df = pd.DataFrame(web)
df.head()
```

Out[237]:

	Programming Language	2022	2017	2012	2007	2002	1997	1992	1987
0	Python	1	5	8	7	12	28	-	-
1	C	2	2	1	2	2	1	1	1
2	Java	3	1	2	1	1	16	-	-
3	C++	4	3	3	3	3	2	2	6
4	C#	5	4	4	8	14	-	-	-

In [238]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14 entries, 0 to 13
Data columns (total 9 columns):
 #   Column               Non-Null Count  Dtype 
---  ------               --------------  ----- 
 0   Programming Language  14 non-null     object
 1   2022                 14 non-null     object
 2   2017                 14 non-null     object
 3   2012                 14 non-null     object
 4   2007                 14 non-null     object
 5   2002                 14 non-null     object
 6   1997                 14 non-null     object
 7   1992                 14 non-null     object
 8   1987                 14 non-null     object
dtypes: object(9)
memory usage: 1.1+ KB
```

to_numeric: Convierte datos texto en números y **apply()** permite aplicar un método a todo el conjunto:

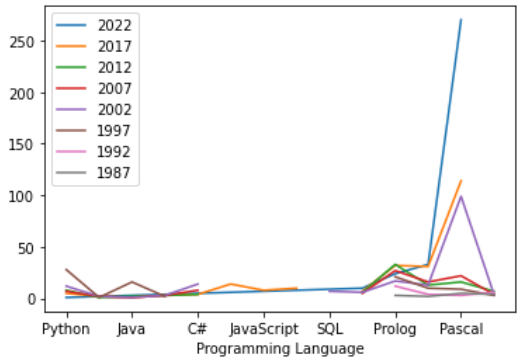
In [239]:
```
02','1997','1992','1987']] = df[['2022','2017','2012', '2007','2002','1997','1992','1987']].apply(pd.to_numeric,errors='coerce')
```

Out[239]:

	Programming Language	2022	2017	2012	2007	2002	1997	1992	1987
0	Python	1.0	5.0	8.0	7.0	12.0	28.0	NaN	NaN
1	C	2.0	2.0	1.0	2.0	2.0	1.0	1.0	1.0
2	Java	3.0	1.0	2.0	1.0	1.0	16.0	NaN	NaN
3	C++	4.0	3.0	3.0	3.0	3.0	2.0	2.0	6.0
4	C#	5.0	4.0	4.0	8.0	14.0	NaN	NaN	NaN

In [240]:
```
df.plot(x='Programming Language')
```

Out[240]: <AxesSubplot:xlabel='Programming Language'>



fillna() convierte los NaN en número:

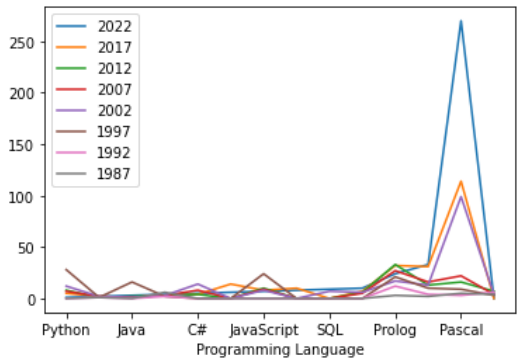
In [241]:
```
df = df.fillna(0)
df.head()
```

Out[241]:

	Programming Language	2022	2017	2012	2007	2002	1997	1992	1987
0	Python	1.0	5.0	8.0	7.0	12.0	28.0	0.0	0.0
1	C	2.0	2.0	1.0	2.0	2.0	1.0	1.0	1.0
2	Java	3.0	1.0	2.0	1.0	1.0	16.0	0.0	0.0
3	C++	4.0	3.0	3.0	3.0	3.0	2.0	2.0	6.0
4	C#	5.0	4.0	4.0	8.0	14.0	0.0	0.0	0.0

In [242]:
```
df.plot.line(x='Programming Language')
```

Out[242]: <AxesSubplot:xlabel='Programming Language'>



Según el caso puede necesitarse instalar alguno de los siguientes módulos:

- **pip install lxml**
- **pip install html5lib**
- **pip install beautifulsoup4**

ExcelFile: permite acceder a los datos almacenados en un archivo xlsx.
sheet_names: permite acceder a los nombres de las hojas del archivo xlsx.

In [243]:
```
xls = pd.ExcelFile('archs/14.autos.xlsx')
print(xls.sheet_names)
```

```
['autos', 'marca']
```

parse: parsea las hoja elegida:

In [244]:
```
autos = xls.parse('autos')
autos.head()
```

Out[244]:

	Orden	IDMARCA	MODELO	TIPO	PRECIO	AUMENTO	STOCK
0	1	100	99 Cavalier	Descapotable	19571	0.06	6
1	2	100	99 Blazer	Deportivo	18470	0.02	5
2	3	100	99 Camaro	Descapotable	22205	0.04	9
3	4	100	99 Malibu	Sedán Familiar	16000	0.06	5
4	5	100	99 Lumina	Sedán Familiar	18190	0.06	8

In [245]:
```
autos = autos.drop(['Orden'], axis=1)
autos.head()
```

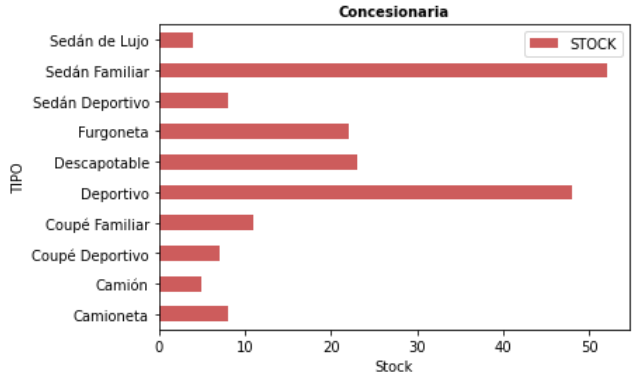
Out[245]:

	IDMARCA	MODELO	TIPO	PRECIO	AUMENTO	STOCK
0	100	99 Cavalier	Descapotable	19571	0.06	6
1	100	99 Blazer	Deportivo	18470	0.02	5
2	100	99 Camaro	Descapotable	22205	0.04	9
3	100	99 Malibu	Sedán Familiar	16000	0.06	5
4	100	99 Lumina	Sedán Familiar	18190	0.06	8

groupby: permite agrupar datos y luego aplicar operaciones:

In [246]:
```
autos.groupby('TIPO')['STOCK'].sum().plot(kind='barh', legend='Reverse', color='indianred')
plt.xlabel('Stock')
plt.title('Concesionaria', weight='bold', size=10)
```

Out[246]:
```
Text(0.5, 1.0, 'Concesionaria')
```



In [247]:
```
marca = xls.parse('marca')
marca.head()
```

Out[247]:

	id	nombre_marca	codigo
0	100	Chevrolet	AA
1	200	Chrysler	AC
2	300	Dodge	AA
3	400	Ford	AE
4	500	GMC	AF

In [248]:
```
mezcla = pd.merge(marca, autos, left_on='id', right_on='IDMARCA')
mezcla.head()
```

Out[248]:

	id	nombre_marca	codigo	IDMARCA	MODELO	TIPO	PRECIO	AUMENTO	STOCK
0	100	Chevrolet	AA	100	99 Cavalier	Descapotable	19571	0.06	6
1	100	Chevrolet	AA	100	99 Blazer	Deportivo	18470	0.02	5
2	100	Chevrolet	AA	100	99 Camaro	Descapotable	22205	0.04	9
3	100	Chevrolet	AA	100	99 Malibu	Sedán Familiar	16000	0.06	5
4	100	Chevrolet	AA	100	99 Lumina	Sedán Familiar	18190	0.06	8

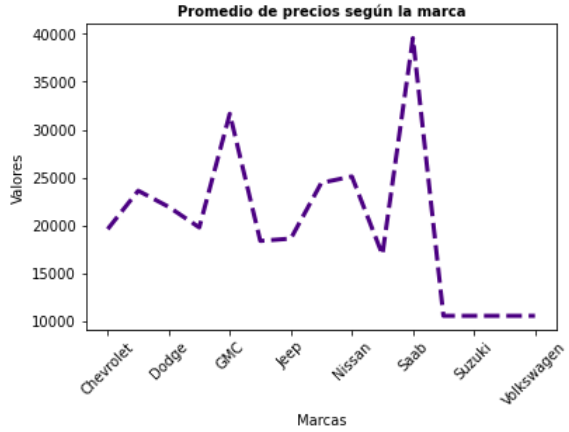
In [249]:
```
mezcla = mezcla.drop(['codigo', 'AUMENTO'], axis=1)
mezcla.head()
```

Out[249]:

	id	nombre_marca	IDMARCA	MODELO	TIPO	PRECIO	STOCK
0	100	Chevrolet	100	99 Cavalier	Descapotable	19571	6
1	100	Chevrolet	100	99 Blazer	Deportivo	18470	5
2	100	Chevrolet	100	99 Camaro	Descapotable	22205	9
3	100	Chevrolet	100	99 Malibu	Sedán Familiar	16000	5
4	100	Chevrolet	100	99 Lumina	Sedán Familiar	18190	8

In [250]:
```
mezcla.groupby('nombre_marca')['PRECIO'].mean().plot(kind='line', rot=45, color='indigo',
                                                     linewidth=3, linestyle='--')
plt.xlabel('Marcas')
plt.ylabel('Valores')
plt.title('Promedio de precios según la marca', weight='bold', size=10)
```

Out[250]: Text(0.5, 1.0, 'Promedio de precios según la marca')



Según el caso puede necesitar instalar los siguientes módulos:

- **pip install xlrd**
- **pip install openpyxl**

In [251]:
```
datos = pd.read_csv('archs/14.comercio_interno.csv', encoding='latin-1')
df = pd.DataFrame(datos)
df.head()
```

Out[251]:

	sector_id	sector_nombre	variable_id	actividad_producto_nombre	indicador	unidad_de_medida	fuente	frecuencia_nombre	cobertura_nombre	alcance_tip
0	31	Comercio interno	330	Alimentos preparados y rotisería	Vtas en super (amp), a precios ctes	miles de pesos	INDEC	Mensual	Nacional	PAI
1	31	Comercio interno	330	Alimentos preparados y rotisería	Vtas en super (amp), a precios ctes	miles de pesos	INDEC	Mensual	Nacional	PAI
2	31	Comercio interno	330	Alimentos preparados y rotisería	Vtas en super (amp), a precios ctes	miles de pesos	INDEC	Mensual	Nacional	PAI
3	31	Comercio interno	330	Alimentos preparados y rotisería	Vtas en super (amp), a precios ctes	miles de pesos	INDEC	Mensual	Nacional	PAI
4	31	Comercio interno	330	Alimentos preparados y rotisería	Vtas en super (amp), a precios ctes	miles de pesos	INDEC	Mensual	Nacional	PAI

Eliminamos filas que no necesitaremos:

In [252]:
```
df = df.drop(df[df['alcance_nombre'] == 'Argentina'].index)
df = df.drop(df[df['alcance_nombre'] == 'GRAN BUENOS AIRES'].index)
df = df.drop(df[df['alcance_nombre'] == 'INDETERMINADA'].index)
df = df.drop(df[df['alcance_nombre'] == 'PARTIDOS DEL GBA'].index)
df = df.drop(df[df['alcance_nombre'] == 'RESTO DE BUENOS AIRES'].index)
```

Eliminamos columnas innecesarias:

In [253]:
```
df = df.drop(['sector_id','sector_nombre','variable_id','indicador','unidad_de_medida','fuente','frecuencia_nombre',
       'cobertura_nombre','alcance_id'], axis=1)

df.head()
```

Out[253]:

	actividad_producto_nombre	alcance_tipo	alcance_nombre	indice_tiempo	valor
33	Alimentos preparados y rotisería	PROVINCIA	CAPITAL FEDERAL	01/01/2017	83483.478
34	Alimentos preparados y rotisería	PROVINCIA	CAPITAL FEDERAL	01/02/2017	82264.716
35	Alimentos preparados y rotisería	PROVINCIA	CAPITAL FEDERAL	01/03/2017	94698.366
36	Alimentos preparados y rotisería	PROVINCIA	CAPITAL FEDERAL	01/04/2017	96251.926
37	Alimentos preparados y rotisería	PROVINCIA	CAPITAL FEDERAL	01/05/2017	90975.933

Aplicamos datetime para poder trabajar con fechas:

In [254]:
```
import datetime

df['indice_tiempo'] = pd.to_datetime(df['indice_tiempo'], format='%d/%m/%Y')
df['año'], df['mes'] = df['indice_tiempo'].dt.year, df['indice_tiempo'].dt.month
df = df.drop(df[df['año'] < 2010].index)
```

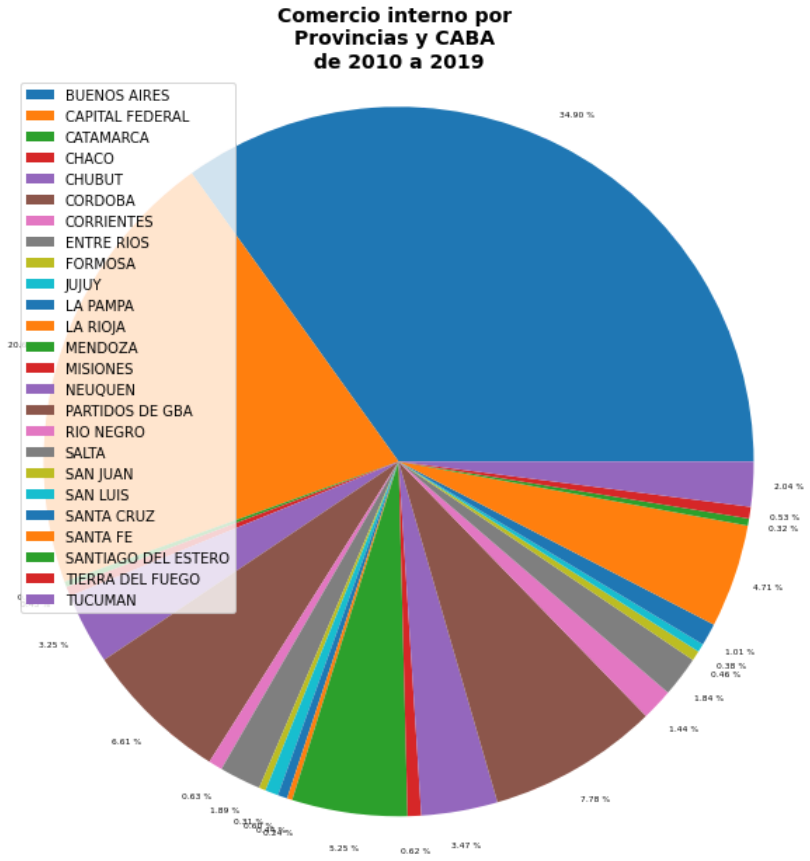
to_csv: permite generar un archivo en formato csv:

In [255]:
```
df.to_csv('archs/comercio-interno-2.csv')
```

Agrupamos los datos y graficamos:

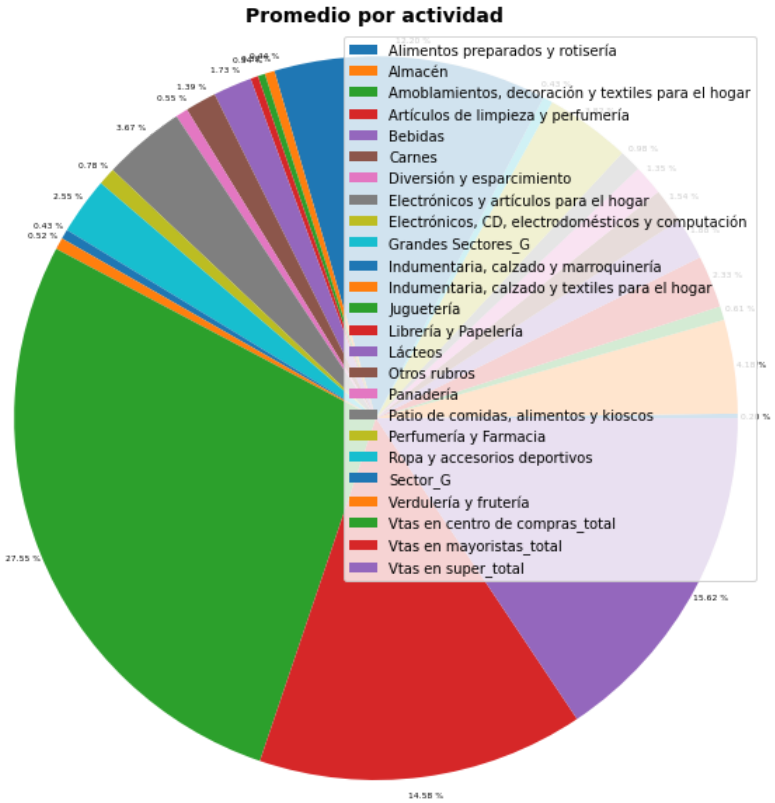
In [256]:
```
fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111)
df.groupby('alcance_nombre')['valor'].sum().plot(kind='pie',
                                                 legend='Reverse',
                                                 autopct='%0.2f %%',
                                                 fontsize=6,
                                                 labels=None,
                                                 pctdistance=1.10
                                                 )
plt.axis('equal')
plt.ylabel('')
plt.tight_layout()
plt.title('Comercio interno por\nProvincias y CABA\n de 2010 a 2019', weight='bold', size=14)
```

Out[256]: Text(0.5, 1.0, 'Comercio interno por\nProvincias y CABA\n de 2010 a 2019')



In [257]:
```
fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111)
df.groupby('actividad_producto_nombre')['valor'].mean().plot(kind='pie',
                                                             legend='Reverse',
                                                             autopct='%0.2f %%',
                                                             fontsize=6,
                                                             labels=None,
                                                             pctdistance=1.05)
plt.axis('equal')
plt.ylabel('')
plt.tight_layout()
plt.title('Promedio por actividad', weight='bold', size=14)
```

Out[257]: Text(0.5, 1.0, 'Promedio por actividad')



read_table: permite leer datos de un archivo txt:

In [258]:
```
userHeader = ['user_id', 'gender', 'age', 'ocupation', 'zip']
users = pd.read_table('archs/dataset/users.txt', engine='python', sep='::', header=None, names=userHeader)
users.head()
```

Out[258]:

	user_id	gender	age	ocupation	zip
0	1	F	1	10	48067
1	2	M	56	16	70072
2	3	M	25	15	55117
3	4	M	45	7	02460
4	5	M	25	20	55455

In [259]:
```
ratingHeader = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('archs/dataset/ratings.txt', engine='python', sep='::', header=None, names=ratingHeader)
ratings.head()
```

Out[259]:

	user_id	movie_id	rating	timestamp
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291

In [260]:
```
mergeRatings = pd.merge(users, ratings, on='user_id')
mergeRatings.head()
```

Out[260]:

	user_id	gender	age	ocupation	zip	movie_id	rating	timestamp
0	1	F	1	10	48067	1193	5	978300760
1	1	F	1	10	48067	661	3	978302109
2	1	F	1	10	48067	914	3	978301968
3	1	F	1	10	48067	3408	4	978300275
4	1	F	1	10	48067	2355	5	978824291

In [261]:
```
mergeRatings = mergeRatings.drop(['user_id','zip','timestamp','ocupation'], axis=1)
mergeRatings.head()
```

Out[261]:

	gender	age	movie_id	rating
0	F	1	1193	5
1	F	1	661	3
2	F	1	914	3
3	F	1	3408	4
4	F	1	2355	5

In [262]:
```
movieHeader = ['movie_id', 'title', 'genders']
movies = pd.read_table('archs/dataset/movies.txt', engine='python', sep='::', header=None,
                       names=movieHeader, encoding='latin-1')
movies.head()
```

Out[262]:

	movie_id	title	genders
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy

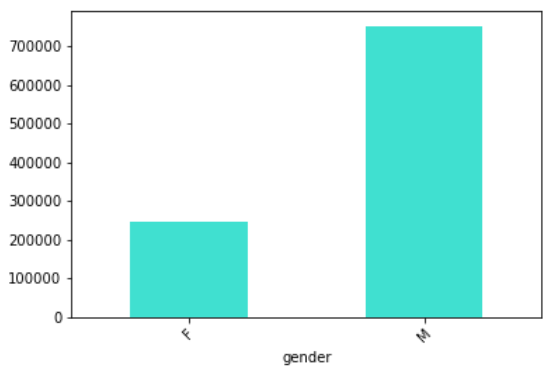
In [263]:
```
merge = pd.merge(mergeRatings, movies)
merge.head()
```

Out[263]:

	gender	age	movie_id	rating	title	genders
0	F	1	1193	5	One Flew Over the Cuckoo's Nest (1975)	Drama
1	M	56	1193	5	One Flew Over the Cuckoo's Nest (1975)	Drama
2	M	25	1193	4	One Flew Over the Cuckoo's Nest (1975)	Drama
3	M	25	1193	4	One Flew Over the Cuckoo's Nest (1975)	Drama
4	M	50	1193	5	One Flew Over the Cuckoo's Nest (1975)	Drama

In [264]:
```
merge.groupby('gender').size().plot(kind='bar', fontsize=10, rot=45, color='turquoise')
```

Out[264]: <AxesSubplot:xlabel='gender'>



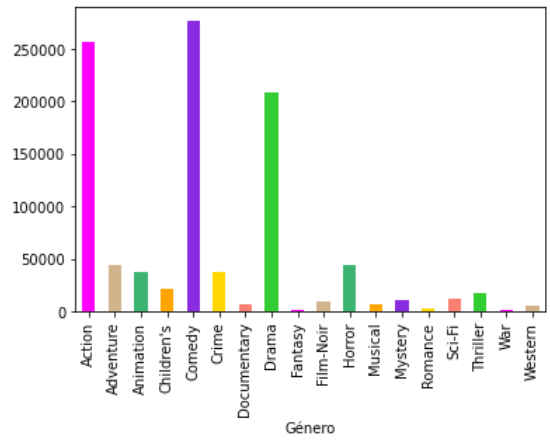
In [265]:
```
merge["Género"] = merge["genders"].str.split('|', n=1, expand= True)[0]
merge.head()
```

Out[265]:

	gender	age	movie_id	rating	title	genders	Género
0	F	1	1193	5	One Flew Over the Cuckoo's Nest (1975)	Drama	Drama
1	M	56	1193	5	One Flew Over the Cuckoo's Nest (1975)	Drama	Drama
2	M	25	1193	4	One Flew Over the Cuckoo's Nest (1975)	Drama	Drama
3	M	25	1193	4	One Flew Over the Cuckoo's Nest (1975)	Drama	Drama
4	M	50	1193	5	One Flew Over the Cuckoo's Nest (1975)	Drama	Drama

In [266]:
```
colors = ['magenta','tan','mediumseagreen','orange','blueviolet', 'gold', 'salmon', 'limegreen']
merge.groupby('Género').size().plot(kind='bar', color=colors)
```

Out[266]: <AxesSubplot:xlabel='Género'>



In [267]:
```
info1000 = merge.loc[1000]
print ('Info de la posición 1000 en la tabla:',info1000)
```

```
Info de la posición 1000 en la tabla: gender                                        M
age                                          25
movie_id                                   1193
rating                                        5
title       One Flew Over the Cuckoo's Nest (1975)
genders                                   Drama
Género                                    Drama
Name: 1000, dtype: object
```

In [268]:
```
info5_97 = merge[5:97]
print ('Info de la posición 5 a la 96 en la tabla::',info5_97)
```

```
Info de la posición 5 a la 96 en la tabla::    gender  age  movie_id  rating                                title  \
5       F   18      1193       4  One Flew Over the Cuckoo's Nest (1975)
6       M    1      1193       5  One Flew Over the Cuckoo's Nest (1975)
7       F   25      1193       5  One Flew Over the Cuckoo's Nest (1975)
8       F   25      1193       3  One Flew Over the Cuckoo's Nest (1975)
9       M   45      1193       5  One Flew Over the Cuckoo's Nest (1975)
..    ...  ...       ...     ...                                   ...
92      F   50      1193       5  One Flew Over the Cuckoo's Nest (1975)
93      M   50      1193       3  One Flew Over the Cuckoo's Nest (1975)
94      M   35      1193       5  One Flew Over the Cuckoo's Nest (1975)
95      M   35      1193       4  One Flew Over the Cuckoo's Nest (1975)
96      M   25      1193       4  One Flew Over the Cuckoo's Nest (1975)

   genders Género
5    Drama  Drama
6    Drama  Drama
7    Drama  Drama
8    Drama  Drama
9    Drama  Drama
..     ...    ...
92   Drama  Drama
93   Drama  Drama
94   Drama  Drama
95   Drama  Drama
96   Drama  Drama

[92 rows x 7 columns]
```

In [269]:
```
numberRatings = merge.groupby('title').size().sort_values(ascending=False)
print ('Primeras 10 películas con más votos:', numberRatings[:10])
```

```
Primeras 10 películas con más votos: title
American Beauty (1999)                                3428
Star Wars: Episode IV - A New Hope (1977)             2991
Star Wars: Episode V - The Empire Strikes Back (1980) 2990
Star Wars: Episode VI - Return of the Jedi (1983)     2883
Jurassic Park (1993)                                  2672
Saving Private Ryan (1998)                            2653
Terminator 2: Judgment Day (1991)                     2649
Matrix, The (1999)                                    2590
Back to the Future (1985)                             2583
Silence of the Lambs, The (1991)                      2578
dtype: int64
```

In [270]:
```
avgRatings = merge.groupby(['movie_id', 'title']).mean()
print ('Media del rating:', avgRatings['rating'][:10])
```

```
Media del rating: movie_id  title
1         Toy Story (1995)                    4.146846
2         Jumanji (1995)                      3.201141
3         Grumpier Old Men (1995)             3.016736
4         Waiting to Exhale (1995)            2.729412
5         Father of the Bride Part II (1995)  3.006757
6         Heat (1995)                         3.878723
7         Sabrina (1995)                      3.410480
8         Tom and Huck (1995)                 3.014706
9         Sudden Death (1995)                 2.656863
10        GoldenEye (1995)                    3.540541
Name: rating, dtype: float64
```

agg: permite aplicar funciones de agregación:

In [271]:
```
dataRatings = merge.groupby(['movie_id', 'title'])['rating'].agg(['mean', 'sum', 'count', 'std'])
print ('Info estadística del rating:', dataRatings[:10])
```

```
Info estadística del rating:                                        mean   sum  count       std
movie_id title
1        Toy Story (1995)                   4.146846  8613   2077  0.852349
2        Jumanji (1995)                     3.201141  2244    701  0.983172
3        Grumpier Old Men (1995)            3.016736  1442    478  1.071712
4        Waiting to Exhale (1995)           2.729412   464    170  1.013381
5        Father of the Bride Part II (1995) 3.006757   890    296  1.025086
6        Heat (1995)                        3.878723  3646    940  0.934588
7        Sabrina (1995)                     3.410480  1562    458  0.979918
8        Tom and Huck (1995)                3.014706   205     68  0.954059
9        Sudden Death (1995)                2.656863   271    102  1.048290
10       GoldenEye (1995)                   3.540541  3144    888  0.891233
```