😎 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, se puede lograr cinco pasos típicos en el procesamiento y análisis de datos, independientemente del origen de los datos: cargar, preparar, manipular, modelar y analizar.

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 el dato es 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 la que se puede acceder con los valores y los atributos de índice. Aquí no se pasa 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    0.142028
1    1.960865
2   -0.309003
3   -0.856571
dtype: float64
```

Diferencia entre el índice de pandas y numpy:

- **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.

Ejercicios:

1. Crear un objeto Serie con 10 elementos aleatorios. Luego, convertir el objeto Series creado, en una lista Python.

2. Aplica las operaciones aritméticas básicas sobre 2 objetos Series.

```
serie1 = pd.Series([1, 2, 3, 4, 5])
serie2 = pd.Series([9, 8, 7, 6, 5])
```

3. Usa operadores relacionales para comparar los objetos Series creados en el anterior.

4. Cambia el tipo de datos de un objeto Series a número.

```
datos = pd.Series(['100', '200', 'python', '300.15', '500.8'])
```

Atributos y funciones

Crear dándole los valores a index

Se puede usar índices no contiguos o no secuenciales:

In [8]:
```
s = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
s
```

Out[8]:
```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

Crear desde un diccionario

In [9]:
```
population_dict = {'Buenos Aires': 8332521,
                   'Córdoba': 7448193,
                   'Mendoza': 1965112,
                   'Neuquén': 1955607,
                   'Santa Fé': 1281353}

s = pd.Series(population_dict)
print (f"Datos:\n{s}\nLos índices son: {s.index}")
```

```
Datos:
Buenos Aires    8332521
Córdoba         7448193
Mendoza         1965112
Neuquén         1955607
Santa Fé        1281353
dtype: int64
Los índices son: Index(['Buenos Aires', 'Córdoba', 'Mendoza', 'Neuquén', 'Santa Fé'], dtype='object')
```

In [10]:
```
print (f"values devuelve los datos reales de la serie como matriz:\n{s.values}\n\n\
items devuelve una tupla iterable (índice, valor):\n{s.items}")
print (f"\naxes devuelve las etiquetas del eje:\n{s.axes}\n\n\
empty devuelve el valor booleano que indica si el objeto está vacío o no.\n\
Verdadero indica que el objeto está vacío:\n{s.empty}")
print(f"\nndim devuelve las dimensiones del objeto, por definición una serie es 1D\n\
por lo que devolverá: {s.ndim}\n\nsize devuelve el tamaño del objeto es: {s.size}")
```

```
values devuelve los datos reales de la serie como matriz:
[8332521 7448193 1965112 1955607 1281353]

items devuelve una tupla iterable (índice, valor):
<bound method Series.items of Buenos Aires    8332521
Córdoba         7448193
Mendoza         1965112
Neuquén         1955607
Santa Fé        1281353
dtype: int64>

axes devuelve las etiquetas del eje:
[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:
False

ndim devuelve las dimensiones del objeto, por definición una serie es 1D
por lo que devolverá: 1

size devuelve el tamaño del objeto es: 5
```

Ejercicio:

1. Construir un programa que pregunte al usuario un rango de días -datos de tipo entero- (el usuario debe ingresar el día inicial y día final), luego pida las ventas de cada día y por último emita por pantalla una serie con los datos de las ventas indexada por los días, antes y después de aplicarles un descuento del 10%.

concat()

In [11]:
```
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[11]:
```
1    A
2    B
3    C
1    D
3    E
5    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 [12]:
```
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 [13]:
```
pd.concat([s1, s2], ignore_index=True)
```

Out[13]:
```
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.

Ejercicios:

1. Escribir una función que reciba un diccionario con las notas de 5 alumnos de un curso
y devuelva una serie con la nota mínima, la máxima, media y la desviación típica.

```
notas = {'Juan':9, 'María':6.5, 'Pedro':4, 'Carmen': 8.5, 'Luis': 5}
```

2. Escribir una función que reciba el diccionario dado en el punto anterior y devuelva una serie
con las notas de los alumnos aprobados ordenadas de mayor a menor.

Acceso a datos de series con posición e índices

Nota: tema ampliado en apunte 'Indexación y Slicing en Pandas.pdf'

Funciones universales: Conservación de índices

Debido a que Pandas está diseñado para funcionar con NumPy, cualquier función universal de NumPy también puede aplicarse en objetos Series y DataFrame.

In [14]:
```
rng = np.random.RandomState(42)
s = pd.Series(rng.randint(0, 10, 4))
s
```

Out[14]:
```
0    6
1    3
2    7
3    4
dtype: int32
```

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

In [15]:
```
# https://stackoverflow.com/questions/31951980/what-exactly-does-numpy-exp-do
res = np.exp(s)
res, type(res)
```

Out[15]:
```
(0     403.428793
 1      20.085537
 2    1096.633158
 3      54.598150
 dtype: float64,
 pandas.core.series.Series)
```

Alineación de índices en Series

Para operaciones binarias de 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 [16]:
```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B
```

Out[16]:
```
0    NaN
1    5.0
2    9.0
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*

In [17]:
```
A.add(B, fill_value=0)
```

Out[17]:
```
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 [18]:
```
pd.Series([1, np.nan, 2, None])
```

Out[18]:
```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

In [19]:
```
x = pd.Series(range(2), dtype=int)
x
```

Out[19]:
```
0    0
1    1
dtype: int32
```

In [20]:
```
x[0] = None
x
```

Out[20]:
```
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:

In [21]:
```
s = pd.Series([1, np.nan, 'hola', None])
s
```

Out[21]:
```
0       1
1     NaN
2    hola
3    None
dtype: object
```

In [22]:
```
print(f"isnull() genera una máscara booleana que indica valores faltantes:\n{s.isnull()}")
print(f"\nnotnull() opuesto a isnull:\n{s[s.notnull()]}")
```

```
isnull() genera una máscara booleana que indica valores faltantes:
0    False
1     True
2    False
3     True
dtype: bool

notnull() opuesto a isnull:
0       1
2    hola
dtype: object
```

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

In [23]:
```
print(f"dropna() devuelve una versión filtrada de los datos:\n{s.dropna()}")
```

```
dropna() devuelve una versión filtrada de los datos:
0       1
2    hola
dtype: object
```

Se pueden completar las entradas de NaN con un solo valor:

In [24]:
```
s = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
s
```

Out[24]:
```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

In [25]:
```
print(f"fillna() devuelve una copia de los datos con los valores faltantes completados o \
imputados:\n{s.fillna(40)}")
print(f"\nSe puede especificar un relleno hacia adelante para propagar el valor anterior hacia \
adelante:\n{s.fillna(method='ffill')}")
print(f"\nSe puede especificar un relleno para propagar los valores hacia atrás:\n{s.fillna(method='bfill')}")
```

```
fillna() devuelve una copia de los datos con los valores faltantes completados o imputados:
a     1.0
b    40.0
c     2.0
d    40.0
e     3.0
dtype: float64

Se puede especificar un relleno hacia adelante para propagar el valor anterior hacia adelante:
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64

Se puede especificar un relleno para propagar los valores hacia atrás:
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 con matrices de datos, es decir que no tenemos que preocuparnos por la dimensión o la forma de la matriz, sino por la operación que queremos que se realice. Para corregir los siguientes datos haríamos:

In [26]:
```
datos = ['peDRO', 'paBlo', None, 'VILMA', 'gUIDO']
nombres = pd.Series(datos)
nombres,  nombres.str.capitalize()
```

Out[26]:
```
(0    peDRO
 1    paBlo
 2     None
 3    VILMA
 4    gUIDO
 dtype: object,
 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 [27]:
```
famosos = pd.Series(['Jack Nicholson', 'Dustin Hoffman', 'Tom Hanks','Johnny Depp',
                     'Anthony Hopkins','Richard Gere'])
```

In [28]:
```
print(f"len():\n{famosos.str.len()}\n\nlower():\n{famosos.str.lower()}\n\nstartswith():\n\
{famosos.str.startswith('T')}\n\nsplit():\n{famosos.str.split()}")
```

```
len():
0    14
1    14
2     9
3    11
4    15
5    12
dtype: int64

lower():
0     jack nicholson
1     dustin hoffman
2          tom hanks
3        johnny depp
4    anthony hopkins
5       richard gere
dtype: object

startswith():
0    False
1    False
2     True
3    False
4    False
5    False
dtype: bool

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

Métodos que permiten otras operaciones.

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
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

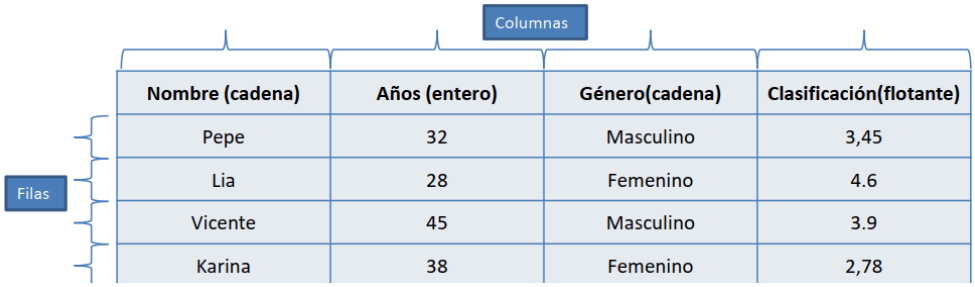
In [29]:
```
print(f"slice() recupera los primeros n caracteres de la matriz:\n{famosos.str.slice(0, 3)}")
print(f"\nLas operaciones get() y slice(), permiten el acceso a elementos vectorizados desde la matriz. \
\nTambién permiten acceder a elementos de arrays devueltos por split():\n \
{famosos.str.split().str.get(-1)} (Recuperar el apellido de cada entrada)")
```

```
slice() recupera los primeros n caracteres de la matriz:
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():
 0    Nicholson
1      Hoffman
2        Hanks
3         Depp
4      Hopkins
5         Gere
dtype: object (Recuperar el apellido de cada entrada)
```

Dataframe

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



Nombre (cadena)	Años (entero)	Género(cadena)	Clasificación(flotante)
Pepe	32	Masculino	3,45
Lia	28	Femenino	4.6
Vicente	45	Masculino	3.9
Karina	38	Femenino	2,78

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 [30]:
```
df = pd.DataFrame()
df
```

Out[30]:

—

Crear un dataframe a partir de listas

In [31]:
```
data = [1,2,3,4,5]
df = pd.DataFrame(data)
df
```

Out[31]:

	0
0	1
1	2
2	3
3	4
4	5

In [32]:
```
type(df)
```

Out[32]:
```
pandas.core.frame.DataFrame
```

In [33]:
```
data = [['Ale',10],['Pepe',12],['Zule',13]]
df = pd.DataFrame(data,columns=['Nombre','Edad'])
df
```

Out[33]:

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

Out[34]:

	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 [35]:
```
data = {'Nombre':['Tomy', 'Juan', 'Silvio', 'Ricky'],
        'Edad':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
df
```

Out[35]:

	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 diccionarios

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 [36]:
```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
df
```

Out[36]:

	a	b	c
0	1	2	NaN
1	5	10	20.0

In [37]:
```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['primera', 'segunda'])
df
```

Out[37]:

	a	b	c
primera	1	2	NaN
segunda	5	10	20.0

Observa que NaN se agrega en las áreas faltantes.

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

In [38]:
```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df1 = pd.DataFrame(data, index=['primera', 'segunda'], columns=['a', 'b'])
df1
```

Out[38]:

	a	b
primera	1	2
segunda	5	10

In [39]:
```
df2 = pd.DataFrame(data, index=['primera', 'segunda'], columns=['a', 'b1'])
df2
```

Out[39]:

	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 [40]:
```
poblacion_dict = {'Buenos Aires':8332521,'Córdoba':7448193,
                  'Mendoza':1965112,'Neuquén':1955607,'Santa Fé':1281353}
s1 = pd.Series(poblacion_dict)
s1
```

Out[40]:
```
Buenos Aires    8332521
Córdoba         7448193
Mendoza         1965112
Neuquén         1955607
Santa Fé        1281353
dtype: int64
```

In [41]:
```
type(s1)
```

Out[41]:
```
pandas.core.series.Series
```

In [42]:
```
pd.DataFrame(s1, columns=['población'])
```

Out[42]:

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

In [43]:
```
type(pd.DataFrame(s1, columns=['población']))
```

Out[43]:
```
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 [44]:
```
poblacion_dict = {'Buenos Aires':8332521,'Córdoba':7448193,
                  'Mendoza':1965112,'Neuquén':1955607,'Santa Fé':1281353}
s1 = pd.Series(poblacion_dict)
s1
```

Out[44]:
```
Buenos Aires    8332521
Córdoba         7448193
Mendoza         1965112
Neuquén         1955607
Santa Fé        1281353
dtype: int64
```

In [45]:
```
area_dict = {'Buenos Aires':423967, 'Córdoba':695662, 'Mendoza':141297,
             'Neuquén':170312, 'Santa Fé':149995}
s2 = pd.Series(area_dict)
s2
```

Out[45]:
```
Buenos Aires    423967
Córdoba         695662
Mendoza         141297
Neuquén         170312
Santa Fé        149995
dtype: int64
```

In [46]:
```
provincias = pd.DataFrame({'población': s1,'área': s2})
provincias
```

Out[46]:

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

In [47]:
```
provincias.index, provincias.columns
```

Out[47]:
```
(Index(['Buenos Aires', 'Córdoba', 'Mendoza', 'Neuquén', 'Santa Fé'], dtype='object'),
 Index(['población', 'área'], dtype='object'))
```

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

In [48]:
```
type(provincias['área'])
```

Out[48]:
```
pandas.core.series.Series
```

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

In [49]:
```
data = {'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(data)
df
```

Out[49]:

	uno	dos
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 fila:

In [50]:
```
pd.DataFrame(np.random.rand(3, 2), columns=['uno', 'dos'], index=['a', 'b', 'c'])
```

Out[50]:

	uno	dos
a	0.693217	0.113449
b	0.214989	0.339901
c	0.721921	0.412430

Crear un a partir de una matriz estructurada

In [51]:
```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
pd.DataFrame(A)
```

Out[51]:

	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.

Ejercicios:

a. Escribir programa que genere y muestre por pantalla un DataFrame con los datos de la tabla siguiente:

Mes	Ventas	Gastos
Enero	305000	220000
Febrero	356000	234000
Marzo	283000	181000
Abril	339000	207000
Mayo	139000	102000
Junio	257000	203000

b. Escribir una función que reciba un DataFrame con el formato del ejercicio anterior,
y devuelva las ventas menos los gastos de cada mes indicado.

Atributos y funciones

index() *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 [52]:
```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

Out[52]:
```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

In [53]:
```
ind[1], ind[::2]
```

Out[53]:
```
(3, 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 [54]:
```
1 ind[1] = 0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [54], 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 [55]:
```
1 indA = pd.Index([1, 3, 5, 7, 9])
2 indB = pd.Index([2, 3, 5, 7, 11])
3 indA.intersection(indB)
```

Out[55]:
```
Int64Index([3, 5, 7], dtype='int64')
```

In [56]:
```
1 indA.union(indB)
```

Out[56]:
```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

In [57]:
```
1 indA.symmetric_difference(indB)
```

Out[57]:
```
Int64Index([1, 2, 9, 11], dtype='int64')
```

" T " transpone la matriz de datos

In [58]:
```
1 d = {'Nombre':pd.Series(['Tomy','Juan','Ricky','Vilma','Silvio','Sara','José']),
2      'Edad':pd.Series([25,26,25,23,30,29,23]),
3      'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}
4
5 df = pd.DataFrame(d)
6 df
```

Out[58]:

	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 [59]:
```
1 df.T
```

Out[59]:

	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 [60]:
```
1 df.describe()
```

Out[60]:

	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

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

In [61]:
```
1 df.describe(include=['object'])
```

Out[61]:

	Nombre
count	7
unique	7
top	Tomy
freq	1

In [62]:
```
1 df.describe(include='all')
```

Out[62]:

	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

In [63]:
```
1 print(f"\ninfo brinda información del dataframe:\n{df.info()}")
2 print(f"\naxes devuelve la lista de etiquetas de eje de fila y etiquetas de eje de columna:\n{df.axes} \
3 \n\ndtypes devuelve el tipo de datos:\n{df.dtypes}")
4 print(f"\nshape devuelve una tupla que representa la dimensión del Data Frame. Tupla (a, b), \
5 donde a representa el número de filas y b representa el número de columnas):\n{df.shape}")
```

```
<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

info brinda información del dataframe:
None

axes devuelve la lista de etiquetas de eje de fila y etiquetas de eje de columna:
[RangeIndex(start=0, stop=7, step=1), Index(['Nombre', 'Edad', 'Rating'], dtype='object')]

dtypes devuelve el tipo de datos:
Nombre     object
Edad        int64
Rating    float64
dtype: object

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):
(7, 3)
```

empty, **ndim** *tienen el mismo significado que en la Series*

In [64]:
```
1 df.empty,df.ndim
```

Out[64]: (False, 2)

In [65]:
```
print(f"size devuelve la cantidad de elementos del DataFrame:\n{df.size}\n\nvalues devuelve los datos del \
DataFrame como un ndarray:\n{df.values}")
print(f"\nAcceso a un elemento del dataframe:\n{df.values[3]}")
print(f"\nhead() 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:\n{df.head(2)}")
print(f"\ntail() devuelve las últimas n filas:\n{df.tail(3)}")
```

```
size devuelve la cantidad de elementos del DataFrame:
21

values devuelve los datos del DataFrame como un ndarray:
[['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]]

Acceso a un elemento del dataframe:
['Vilma' 23 2.56]

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:
  Nombre  Edad  Rating
0   Tomy    25    4.23
1   Juan    26    3.24

tail() devuelve las últimas n filas:
   Nombre  Edad  Rating
4  Silvio    30     3.2
5    Sara    29     4.6
6    José    23     3.8
```

Agregar y eliminar columnas

In [66]:
```
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[66]:

	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 [67]:
```
df['tres']=pd.Series([10,20,30],index=['a','b','c'])
df
```

Out[67]:

	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 resultado de una operación usando las columnas existentes en el dataframe:

In [68]:
```
df['cuatro'] = df['uno'] + df['tres']
df
```

Out[68]:

	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

del *elimina una columna y no devuelve la columna eliminada:*

In [69]:
```
del df['uno']
df
```

Out[69]:

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

pop() *elimina una columna y devuelve la columna eliminada:*

In [70]:
```
df.pop('tres')
```

Out[70]:
```
a    10.0
b    20.0
c    30.0
d     NaN
Name: tres, dtype: float64
```

Agregar y eliminar filas

append() agregará las filas al final.

In [71]:
```
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[71]:

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

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

In [72]:
```
df = df.drop(0)
df
```

Out[72]:

	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

Nota: tema ampliado en 'Indexación y Slicing en Pandas.pdf'

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

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

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

In [74]:
```
provincias[provincias['área'] > 300000]
```

Out[74]:

	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 [75]:
```
provincias['densidad'] = provincias['población'] / provincias['área']
provincias
```

Out[75]:

	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 [76]:
```
print(f"Acceso a todos los elementos del objeto:\n{provincias.values}")
print(f"\nAcceso a un elemento del objeto:\n{provincias.values[2]}")
```

```
Acceso a todos los elementos del objeto:
[[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]]

Acceso a un elemento del objeto:
[1.96511200e+06 1.41297000e+05 1.39076697e+01]
```

In [77]:
```
provincias.loc[provincias['población'] > 3000000, ['población', 'área']]
```

Out[77]:

	población	área
Buenos Aires	8332521	423967
Córdoba	7448193	695662

Funciones universales: Conservación de índices

Debido a que Pandas está diseñado para funcionar con NumPy, cualquier función universal de NumPy puede aplicarse en Pandas Series y DataFrame.

In [78]:
```
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[78]:

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

In [79]:
```
np.sin(df * np.pi / 4)
```

Out[79]:

	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 [80]:
```
type(np.sin(df * np.pi / 4))
```

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

Funciones universales: Alineación de índices

In [81]:
```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),columns=list('AB'))
A
```

Out[81]:

	A	B
0	1	11
1	5	1

In [82]:
```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),columns=list('BAC'))
B
```

Out[82]:

	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 [83]:
```
A + B
```

Out[83]:

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

Ejercicio:

Investigar: stack(), unstack() y pivot() y construir un ejemplo de cada uno.

Mapeo entre operadores de Python y métodos de Pandas

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

Funciones universales: 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 [84]:
```
A = rng.randint(10, size=(3, 4))
A
```

Out[84]:
```
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 [85]:
```
A - A[0]
```

Out[85]:
```
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 [86]:
```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]
```

Out[86]:

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

In [87]:
```
df.subtract(df['R'], axis=0)
```

Out[87]:

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

In [88]:
```
x = df.iloc[0, ::2]
x
```

Out[88]:
```
Q    3
S    2
Name: 0, dtype: int32
```

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

In [89]:
```
df - x
```

Out[89]:

	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 [90]:
```
df = pd.DataFrame([[1, np.nan, 2], [2, 3, 5], [np.nan, 4, 6]])
df
```

Out[90]:

	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 [91]:
```
df.dropna()
```

Out[91]:

	0	1	2
1	2.0	3.0	5

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

In [92]:
```
df.dropna(axis='columns')
```

Out[92]:

	2
0	2
1	5
2	6

In [93]:
```
df.dropna(axis=1)
```

Out[93]:

	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 [94]:
```
df[3] = np.nan
df
```

Out[94]:

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

In [95]:
```
df.dropna(axis='columns', how='all')
```

Out[95]:

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

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

In [96]:
```
df
```

Out[96]:

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

In [97]:
```
df.dropna(axis='rows', thresh=3)
```

Out[97]:

	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 [98]:
```
df.fillna(method='ffill', axis=1)
```

Out[98]:

	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 [99]:
```
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 [100]:
```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
df1,df2
```

Out[100]:
```
(    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 [101]:
```
pd.concat([df1, df2])
```

Out[101]:

	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 [102]:
```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
pd.concat([df3, df4], axis='columns')
```

Out[102]:

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

*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 [103]:
```
df1 = make_df('ABC', [1, 2])
df2 = make_df('BCD', [3, 4])
df=pd.concat([df1, df2])
df
```

Out[103]:

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

In [104]:
```
df = pd.concat([df1, df2], join='inner')
df
```

Out[104]:

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

merge(): *combinación de conjuntos de datos*

In [105]:
```
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[105]:

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

In [106]:
```
df2
```

Out[106]:

	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 [107]:
```
df = pd.merge(df1, df2)
df
```

Out[107]:

	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 [108]:
```
df3 = pd.DataFrame({'grupo': ['Contabilidad', 'Ingeniería', 'RRHH'],'supervisor': ['Carly', 'Guido', 'Esteban']})
df3
```

Out[108]:

	grupo	supervisor
0	Contabilidad	Carly
1	Ingeniería	Guido
2	RRHH	Esteban

In [109]:
```
pd.merge(df, df3)
```

Out[109]:

	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 [110]:
```
df4 = pd.DataFrame({'grupo': ['Contabilidad', 'Contabilidad', 'Ingeniería', 'Ingeniería', 'RRHH', 'RRHH'],
                    'habilidades': ['matemáticas', 'hojas de cálculo', 'codificación', 'linux','hojas de cálculo',
                                    'organización']})
df4
```

Out[110]:

	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 [111]:
```
pd.merge(df1, df4)
```

Out[111]:

	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 [112]:
```
pd.merge(df1, df2, on='empleado')
```

Out[112]:

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

'left_on' y 'right_on':

In [113]:
```
df5 = pd.DataFrame({'nombre': ['Bob', 'Juan', 'Lisa', 'Susana'],'salario': [70000, 80000, 120000, 90000]})
df5
```

Out[113]:

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

In [114]:
```
pd.merge(df1, df5, left_on="empleado", right_on="nombre")
```

Out[114]:

	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 [115]:
```
pd.merge(df1, df5, left_on="empleado", right_on="nombre").drop('nombre', axis=1)
```

Out[115]:

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

Fusión por índice:

In [116]:
```
df1a = df1.set_index('empleado')
df2a = df2.set_index('empleado')
pd.merge(df1a, df2a, left_index=True, right_index=True)
```

Out[116]:

	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 [117]:
```
df1a.join(df2a)
```

Out[117]:

	grupo	fecha_de_contrato
empleado		
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 [118]:
```
pd.merge(df1a, df5, left_index=True, right_on='nombre')
```

Out[118]:

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

In [119]:
```
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[119]:

	nombre	comida
0	Pedro	pescado
1	Pablo	fruta
2	Mary	pan

In [120]:
```
df2
```

Out[120]:

	nombre	bebida
0	Mary	agua
1	Jose	gaseosa

In [121]:
```
pd.merge(df1, df2)
```

Out[121]:

	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 [122]:
```
pd.merge(df1, df2, how='inner')
```

Out[122]:

	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 [123]:
```
pd.merge(df1, df2, how='outer')
```

Out[123]:

	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 [124]:
```
pd.merge(df1, df2, how='left')
```

Out[124]:

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

In [125]:
```
pd.merge(df1, df2, how='right')
```

Out[125]:

	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 [126]:
```
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[126]:

	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 [127]:
```
df.sum()
```

Out[127]:
```
Nombre    TomyJuanRickyVilmaSilvioSaraJosé
Edad                                   181
Rating                               25.61
dtype: object
```

Suma del eje 1

In [128]:
```
df.sum(1)
```

Out[128]:
```
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 [129]:
```
df.mean(numeric_only=True)
```

Out[129]:
```
Edad      25.857143
Rating     3.658571
dtype: float64
```

std

In [130]:
```
df.std(numeric_only=True)
```

Out[130]:
```
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 [131]:
```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'], 'data': range(6)}, columns=['key', 'data'])
df
```

Out[131]:

	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 [132]:
```
df.groupby('key')
```

Out[132]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000228C3EC53D0>
```

In [133]:
```
df.groupby('key').sum()
```

Out[133]:

key	data
A	3
B	5
C	7

aggregate() , **agg()**: *El método de agregación() permite calcular todos los agregados a la vez:*

In [134]:
```
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[134]:

	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 [135]:
```
df.groupby('key').aggregate(['min', np.median, max])
```

Out[135]:

	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 [136]:
```
df.groupby('key').aggregate({'data1': 'min', 'data2': 'max'})
```

Out[136]:

	data1	data2
key		
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.*

In [137]:
```
df
```

Out[137]:

	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 [138]:
```
df.groupby('key').std()
```

Out[138]:

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

Aquí, como A no supera el valor dado, elimina la fila.

In [139]:
```
def filter_func(x):
    return x['data2'].std() > 4

df.groupby('key').filter(filter_func)
```

Out[139]:

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

transform(): *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 [140]:
```
df
```

Out[140]:

	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 [141]:
```
df.groupby('key').transform(lambda x: x - x.mean())
```

Out[141]:

	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(): *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 [142]:
```
def norm_by_data2(x):
    x['data1'] /= x['data2'].sum()
    return x

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

Out[142]:

	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 [143]:
```
date = pd.to_datetime("4th of May, 2022")
date
```

Out[143]:
```
Timestamp('2022-05-04 00:00:00')
```

In [144]:
```
date.strftime('%A')
```

Out[144]:
```
'Wednesday'
```

Pueden hacerse operaciones vectorizadas directamente en este mismo objeto:

In [145]:
```
date + pd.to_timedelta(np.arange(12), 'D')
```

Out[145]:
```
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 [146]:
```
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[146]:
```
2022-07-04    0
2022-08-04    1
2021-07-04    2
2021-08-04    3
dtype: int64
```

In [147]:
```
data['2021-08-04': '2022-07-04']
```

Out[147]:
```
2022-07-04    0
2021-08-04    3
dtype: int64
```

In [148]:
```
data['2021']
```

Out[148]:
```
2021-07-04    2
2021-08-04    3
dtype: int64
```

In [149]:
```
pd.date_range('2022-07-03', '2022-07-10')
```

Out[149]:
```
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 [150]:
```
import matplotlib.pyplot as plt
import pandas as pd
```

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

In [151]:
```
url='https://www.tiobe.com/tiobe-index/'
web=pd.read_html(url, header=0)[0]
df = pd.DataFrame(web)
df.head(3)
```

Out[151]:

	Mar 2024	Mar 2023	Change	Programming Language	Programming Language.1	Ratings	Change.1
0	1	1	NaN	NaN	Python	15.63%	+0.80%
1	2	2	NaN	NaN	C	11.17%	-3.56%
2	3	4	NaN	NaN	C++	10.70%	-2.59%

In [152]:
```
df = df.drop(['Mar 2024', 'Mar 2023', 'Change','Programming Language', 'Change.1'], axis=1)
df.head()
```

Out[152]:

	Programming Language.1	Ratings
0	Python	15.63%
1	C	11.17%
2	C++	10.70%
3	Java	8.95%
4	C#	7.54%

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

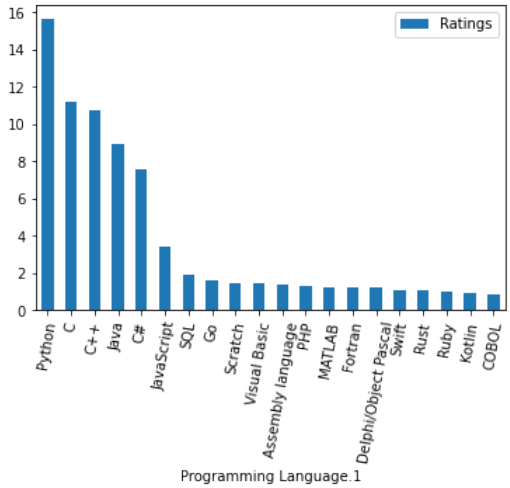
In [153]:
```
df["Ratings"] = df["Ratings"].str.rstrip('%').astype('float')
df.set_index('Programming Language.1',inplace=True)
df.head()
```

Out[153]:

	Ratings
Programming Language.1	
Python	15.63
C	11.17
C++	10.70
Java	8.95
C#	7.54

In [154]:
```
df.plot.bar(rot=80)
```

Out[154]: <AxesSubplot:xlabel='Programming Language.1'>



In [155]:
```
web=pd.read_html(url, header=0)[2]
df = pd.DataFrame(web)
df.head()
```

Out[155]:

	Programming Language	2024	2019	2014	2009	2004	1999	1994	1989
0	Python	1	4	8	6	11	31	22	-
1	C	2	2	1	2	2	1	1	1
2	C++	3	3	4	3	3	2	2	3
3	Java	4	1	2	1	1	12	-	-
4	C#	5	6	5	8	8	29	-	-

to_numeric: *convierte datos texto en números:*

In [156]:
```
df[['2024','2019','2014','2009','2004','1999','1994','1989']] = \
df[['2024','2019','2014','2009','2004','1999','1994','1989']].apply(pd.to_numeric,errors='coerce')
df.head(2)
```

Out[156]:

	Programming Language	2024	2019	2014	2009	2004	1999	1994	1989
0	Python	1.0	4.0	8.0	6.0	11.0	31.0	22.0	NaN
1	C	2.0	2.0	1.0	2.0	2.0	1.0	1.0	1.0

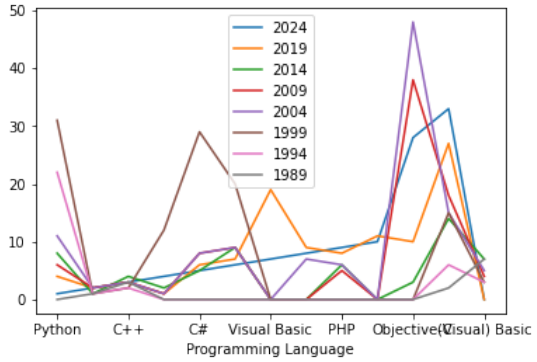
In [157]:
```
df = df.fillna(0)
df.head(2)
```

Out[157]:

	Programming Language	2024	2019	2014	2009	2004	1999	1994	1989
0	Python	1.0	4.0	8.0	6.0	11.0	31.0	22.0	0.0
1	C	2.0	2.0	1.0	2.0	2.0	1.0	1.0	1.0

In [158]:
```
df.plot.line(x='Programming Language')
```

Out[158]: <AxesSubplot:xlabel='Programming Language'>



Según el caso puede necesitarse instalar alguno de los siguientes módulos: lxml, html5lib, 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 [159]:
```
xls = pd.ExcelFile('archs/autos.xlsx')
print(xls.sheet_names)
```

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

parse(): *parsea las hojas elegidas:*

In [160]:
```
autos = xls.parse('autos')
autos.head(2)
```

Out[160]:

	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

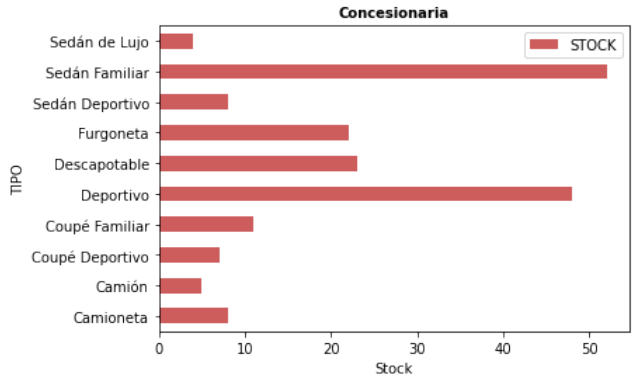
In [161]:
```
autos = autos.drop(['Orden'], axis=1)
autos.head(2)
```

Out[161]:

	IDMARCA	MODELO	TIPO	PRECIO	AUMENTO	STOCK
0	100	99 Cavalier	Descapotable	19571	0.06	6
1	100	99 Blazer	Deportivo	18470	0.02	5

In [162]:
```
autos.groupby('TIPO')['STOCK'].sum().plot(kind='barh', legend='Reverse', color='indianred')
plt.xlabel('Stock')
plt.title('Concesionaria', weight='bold', size=10)
```

Out[162]:
```
Text(0.5, 1.0, 'Concesionaria')
```



In [163]:
```
marca = xls.parse('marca')
marca.head(2)
```

Out[163]:

	id	nombre_marca	codigo
0	100	Chevrolet	AA
1	200	Chrysler	AC

In [164]:
```
mezcla = pd.merge(marca, autos, left_on='id', right_on='IDMARCA').drop('IDMARCA', axis=1)
mezcla.head(2)
```

Out[164]:

	id	nombre_marca	codigo	MODELO	TIPO	PRECIO	AUMENTO	STOCK
0	100	Chevrolet	AA	99 Cavalier	Descapotable	19571	0.06	6
1	100	Chevrolet	AA	99 Blazer	Deportivo	18470	0.02	5

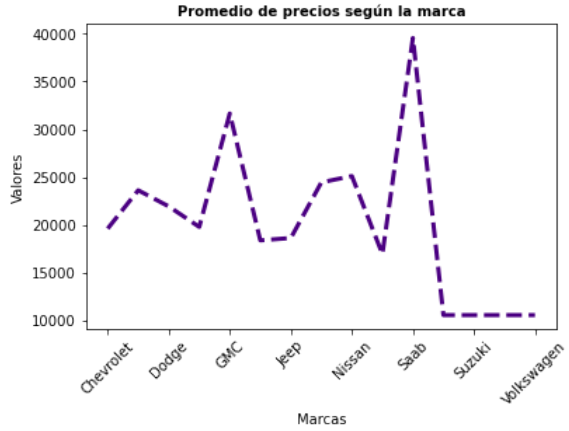
In [165]:
```
mezcla = mezcla.drop(['codigo', 'AUMENTO'], axis=1)
mezcla.head(2)
```

Out[165]:

	id	nombre_marca	MODELO	TIPO	PRECIO	STOCK
0	100	Chevrolet	99 Cavalier	Descapotable	19571	6
1	100	Chevrolet	99 Blazer	Deportivo	18470	5

In [166]:
```
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[166]: Text(0.5, 1.0, 'Promedio de precios según la marca')



Según el caso puede necesitar instalar los siguientes módulos: xlrd, openpyxl

In [167]:
```
datos = pd.read_csv('archs/comercio_interno.csv', encoding='latin-1')
df = pd.DataFrame(datos)
df.head(2)
```

Out[167]:

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

Eliminamos filas que no necesitaremos:

In [168]:
```
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 [169]:
```
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[169]:

	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 [170]:
```
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 [171]:
```
df.to_csv('archs/comercio-interno-2.csv')
```

Agrupamos los datos y graficamos:

In [172]:
```
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[172]: Text(0.5, 1.0, 'Comercio interno por\nProvincias y CABA\n de 2010 a 2019')