Pandas

January 21, 2026

INSTITUTO SUPERIOR TECNOLÓGICO QUITO
Formamos tu PROPÓSITO DE VIDA

1 Ficheros

1.1 Nombre: *Adriel Bedoya*

1.2 Pandas

Es una librería para estructurar datos tabulares, es multivariable (string, bool) y tiene dos clases: **Series** que es de una dimensión (1D) y **DataFrames** que tiene 2 dimensiones (2D)

[2]:
```
# Libería externa
import pandas as pd
```

```
from pandas import Series, DataFrame
```

[3]:
```
#Para ver la versión
pd.__version__
```

[3]: '2.3.3'

1.3 Series

Tiene datos unidimensionales (una sola dimensión), tiene elementos y se pueden modificar sus índices

[4]:
```
#Así creamos una serie con Pandas
paises = pd.Series(["España", "Andorra", "Portugal", "Francia", "Perú"])
print(paises)
```

```
0      España
1     Andorra
2    Portugal
3     Francia
4        Perú
dtype: object
```

[8]:
```
#Especificamos el índice
#Imprime en el orden del 10 al 50 porque tiene salto de 10
paises = pd.Series(["España", "Andorra", "Portugal", "Francia", "Perú"], index␣
 ↪= range(10,60,10))
print(paises)
```

```
10      España
20     Andorra
30    Portugal
40     Francia
50        Perú
dtype: object
```

[13]:
```
#Los índices también pueden ser de más tipos
ciudades_fuchibol = pd.Series(["Quito", "Guayaquil", "Cuenca", "Manabí"],␣
 ↪index=["a","b","c","d"])
print(ciudades_fuchibol)
```

```
a        Quito
b    Guayaquil
c       Cuenca
d       Manabí
dtype: object
```

[15]:
```
#Ahora con atributos
```

```
ciudades_fuchibol.name = "Ciudades con 5 equipos en primera división" #Con esto␣
 ↪ponemos un encabezado a los elementos
ciudades_fuchibol.index.name = "Identificador" #Con esto ponemos encabezado a␣
 ↪los índices
print(ciudades_fuchibol)
```

```
Identificador
a        Quito
b    Guayaquil
c       Cuenca
d       Manabí
Name: Ciudades con 5 equipos en primera división, dtype: object
```

[16]:
```
#El acceso es similar a Numpy o a las listas, según su posición
print(ciudades_fuchibol[1])
```

```
Guayaquil

C:\Users\Usuario-PC\AppData\Local\Temp\ipykernel_28472\618042057.py:2:
FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a
future version, integer keys will always be treated as labels (consistent with
DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
  print(ciudades_fuchibol[1])
```

[27]:
```
#Actualmente se utiliza array.iloc[posición]
print(ciudades_fuchibol.iloc[2])
```

```
Cuenca
```

[29]:
```
#Acceso a través del índice semántico
print(ciudades_fuchibol['c']) #Por aquí si se puede hacer como las listas o␣
 ↪arrays

print(ciudades_fuchibol['c']) == ciudades_fuchibol.iloc[2]
```

```
Cuenca
Cuenca
```

[29]:
```
False
```

1.4 Tratamiento Similar a Ndarray

[34]:
```
#Múltiple recolección de elementos
print(ciudades_fuchibol[['a', 'c']])
print("----------------------------------------")
print(ciudades_fuchibol.iloc[[0,3]])
```

```
Identificador
a     Quito
c    Cuenca
```

```
Name: Ciudades con 5 equipos en primera división, dtype: object
----------------------------------------
Identificador
a     Quito
d    Manabí
Name: Ciudades con 5 equipos en primera división, dtype: object
```

[37]:
```
#Ahora con slicing
#Incluye ambos extremos inicio y fin con el índice semántico
print(ciudades_fuchibol[:'c'])

print("----------------------------------------")

#Aquí le decimos de la posición 0 a la 2
print(ciudades_fuchibol[:2])
```

```
Identificador
a        Quito
b    Guayaquil
c       Cuenca
Name: Ciudades con 5 equipos en primera división, dtype: object
----------------------------------------
Identificador
a        Quito
b    Guayaquil
Name: Ciudades con 5 equipos en primera división, dtype: object
```

[38]:
```
#Casteo a Lista

#Básicamente haremos un parse desde el índice a hasta la c
lista_ejemplo = list(ciudades_fuchibol[:'c'])
print(lista_ejemplo)
print(type(lista_ejemplo))
```

```
['Quito', 'Guayaquil', 'Cuenca']
<class 'list'>
```

[39]:
```
#Recordemos que es una Serie de Pandas
type(ciudades_fuchibol[:'c'])
```

[39]:
```
pandas.core.series.Series
```

[40]:
```
#Cast a un array de numpy

import numpy as np
#Convertimos la Serie en un array que almacenará en la variable 'ciudades'
ciudades = np.array(ciudades_fuchibol[:'c'])
print(ciudades)
print(type(ciudades))
```

```
['Quito' 'Guayaquil' 'Cuenca']
<class 'numpy.ndarray'>
```

[42]:
```
#Casteo a Diccionario
lista = dict(ciudades_fuchibol[:'c'])
print(lista)
```

```
{'a': 'Quito', 'b': 'Guayaquil', 'c': 'Cuenca'}
```

[48]:
```
#Uso para mask para seleccionar

#Creamos una serie que tenga dentro fibonacci
fibonacci = pd.Series([0, 1, 1, 2, 3, 5, 8, 13, 21])
print("primero: ", fibonacci)

print("---------------")
#Creamos una máscara que tenga la condición que vea cuales son mayores a 10
#En bool
mask = fibonacci > 10
print("segundo: ",mask)

print("---------------")
#Con esto, podremos ver los números dentro de la Serie en vez de el valor␣
 ↪boolean
print("tercero: ", fibonacci[mask])
print("---------------")
#Creamos una serie nueva
dst = pd.Series([13,21])
print("cuarto: ", dst)
print("---------------")
#Decimos si "dst" es igual a fibonacci cosa que no pasa porque hay más indices
dst.equals(fibonacci)
#Guardamos en esta variable los números que son mayores a 10
fb = fibonacci[mask]
#Reseteamos los index
#Drop reemplaza los índices por los default (0,1,2,3)
#inplace modifica la serie directamente
fb.reset_index(drop=True, inplace=True)

print(fb)#print
#Ahora si saldrá true porque tenemos los mismos valores e índices
dst.equals(fb)
```

```
primero:  0     0
1     1
2     1
3     2
4     3
5     5
```

```
6     8
7    13
8    21
dtype: int64
---------------
segundo:  0    False
1    False
2    False
3    False
4    False
5    False
6    False
7     True
8     True
dtype: bool
---------------
tercero:  7    13
8    21
dtype: int64
---------------
cuarto:  0    13
1    21
dtype: int64
---------------
0    13
1    21
dtype: int64
```

[48]:
```
True
```

[49]:
```
#Aplicar funciones de Numpy a la Serie
import numpy as np
#Lo que hacemos acá es sumar todos los números dentro de fibonacci
print(np.sum(fibonacci))
```

```
54
```

[50]:
```
#filtrado con np.where
#Creamos una serie con valores y el nan significa que no tien valor
distances = pd.Series([12.1,np.nan,12.8,76.9,6.1,7.2])
#Abajo haremos una comprobación donde veremos que todo nan devuelva el número 2
#pd.notnull() = devuelve True si el valor no es NaN y devuelve false si lo es
#el np.where funciona como un if vectorizado en el cual
#Si no es NaN
valid_distances = np.where(pd.notnull(distances),distances,2)
print(valid_distances)
print(type(valid_distances))
```

```
[12.1  2.  12.8 76.9  6.1  7.2]
```

```
<class 'numpy.ndarray'>
```

1.4.1 Iteración

[51]:
```
#iterar sobre elementos
for value in fibonacci:
    print('Value: ' + str(value))
```

```
Value: 0
Value: 1
Value: 1
Value: 2
Value: 3
Value: 5
Value: 8
Value: 13
Value: 21
```

[52]:
```
# iterar sobre indices
for index in fibonacci.index:
    print('Index: ' + str(index))
```

```
Index: 0
Index: 1
Index: 2
Index: 3
Index: 4
Index: 5
Index: 6
Index: 7
Index: 8
```

[53]:
```
# iterar sobre elementos e índices al mismo tiempo mediante el método de pandas
for index, value in fibonacci.items():
    print('Index: ' + str(index) + ' Value: ' + str(value))
```

```
Index: 0 Value: 0
Index: 1 Value: 1
Index: 2 Value: 1
Index: 3 Value: 2
Index: 4 Value: 3
Index: 5 Value: 5
Index: 6 Value: 8
Index: 7 Value: 13
Index: 8 Value: 21
```

[54]:
```
#Hace lo mismo que el código de arriba pero usando zip
for index, value in zip(fibonacci.index, fibonacci):
    print('Index: ' + str(index) + ' Value: ' + str(value))
```

```
Index: 0 Value: 0
Index: 1 Value: 1
Index: 2 Value: 1
Index: 3 Value: 2
Index: 4 Value: 3
Index: 5 Value: 5
Index: 6 Value: 8
Index: 7 Value: 13
Index: 8 Value: 21
```

1.4.2 Series en Diccionarios

Interpreta el índice como una clave y acepta operaciones para diccionarios

[59]:
```
# crear una serie a partir de un diccionario

#Creamos una serie
serie = pd.Series( { 'Carlos' : 100, 'Marcos': 98} )
print("---------------------------")
#Vemos los índices
print(serie.index)
print("---------------------------")
#Vemos los valores
print(serie.values)
print("---------------------------")
#print de valores completos
print(serie)
print("---------------------------")
print(type(serie))
```

```
---------------------------
Index(['Carlos', 'Marcos'], dtype='object')
---------------------------
[100  98]
---------------------------
Carlos    100
Marcos     98
dtype: int64
---------------------------
<class 'pandas.core.series.Series'>
```

[60]:
```
# añade y elimina elementos a través de índices
serie['Pedro'] = 12
serie['Pedro']=15
del serie['Marcos']
print(serie)
```

```
Carlos    100
Pedro      15
```

```
dtype: int64
```

[64]:
```
# query una serie

#print(serie['Marcos'])
print("-----------------------------------")
#Si Marcos está en la serie
if 'Marcos' in serie:
    #imprime los valores de la serie de Marcos
    print(serie['Marcos'])

print(serie)
```

```
-----------------------------------
Carlos    100
Pedro      15
dtype: int64
```

1.4.3 Operadores entre Series

[66]:
```
# suma de dos series
# Los índices se suman de manera uniforme, si sobra o falta un índice, ese se␣
 ↪hace NaN
serie1 = pd.Series([10,20,30,40], index=range(4) )
serie2 = pd.Series([1,2,3], index=range(3) ) #este no tiene índice 3, por lo␣
 ↪cual, se hace NaN
suma = serie1 + serie2
print(suma)
```

```
0    11.0
1    22.0
2    33.0
3     NaN
dtype: float64
```

[67]:
```
# resta de series (similar a la suma)
print(serie1 - serie2)
```

```
0     9.0
1    18.0
2    27.0
3     NaN
dtype: float64
```

[68]:
```
# operaciones de pre-filtrado
result = serie1 + serie2
#Si es nulo, entonces el valor será 0
result[pd.isnull(result)] = 0 # mask con isnull()
print(result)
```

```
0    11.0
1    22.0
2    33.0
3     0.0
dtype: float64
```

Diferencias entre Pandas Series y Diccionarios Diccionario es una estructura que relaciona clave y valor de forma arbitraria, por otro lado, Series es una estructura de forma estricta de listas de valores con listas de índice asignado en la posición.

Las Series son más eficientes para ciertas operaciones que los diccionarios, también los valores de entrada pueden ser listas o Arrays de Numpy.

En las Series, los índices semánticos pueden ser enteros o caracteres, en los valores pasa de la misma manera. Además, se podría entender entre una lista y un diccionario Python, pero sería de manera unidimensional (1D)

1.5 DataFrame

Son datos tabulares, es decir, que se basan en filas y columnas. Las columnas son Series con índices compartidos

[70]:
```
#Crear un DataFrame a partir de un diccionario de elementos de la misma␣
 ↪longuitud
diccionario = {
    "Nombre": ["Marisa", "Laura", "Manuel"],
    "Edad": [34,29,12]
}
print(diccionario)

#Las claves se identifican como columnas (los encabezados)
frame = pd.DataFrame(diccionario)
display(frame)
```

```
{'Nombre': ['Marisa', 'Laura', 'Manuel'], 'Edad': [34, 29, 12]}
```

	Nombre	Edad
0	Marisa	34
1	Laura	29
2	Manuel	12

[72]:
```
diccionario = {
    "Nombre": ["Marisa", "Yudi", "Carmilla"],
    "Edad": [34,29,12]
}

#Aquí cambiamos a índices semánticos
frame = pd.DataFrame(diccionario, index = ['a','b','c'])
display(frame)
```

```
     Nombre  Edad
a    Marisa    34
b      Yudi    29
c  Carmilla    12
```

[79]:
```
#Además del index, el parámetro 'columns' especifica el número y orden de las␣
 ↪columnas
frame = pd.DataFrame(
    diccionario, columns = [
        'Nacionalidad','Nombre','Edad','Profesion','Genero'
    ]
)

display(frame)
```

```
  Nacionalidad    Nombre  Edad Profesion Genero
0          NaN    Marisa    34       NaN    NaN
1          NaN      Yudi    29       NaN    NaN
2          NaN  Carmilla    12       NaN    NaN
```

[76]:
```
#Acceso a columnas
nombres = frame['Nombre'] #como en el diccionario
display(nombres)
print(type(nombres))
print("-------------------------------------")
edades = frame['Edad']
display(edades)
print(type(edades))
```

```
0      Marisa
1        Yudi
2    Carmilla
Name: Nombre, dtype: object

<class 'pandas.core.series.Series'>
-------------------------------------

0    34
1    29
2    12
Name: Edad, dtype: int64

<class 'pandas.core.series.Series'>
```

[82]:
```
#Siempre que el nombre de la columna lo permita (como espacios, comas, etc)
#Para que esto funcione no debe tener
#No tenga espacios
#No tenga comas
#No empiece por números
#No coincida con métodos de Pandas (sum, mean, etc.)
```

```
nombres = frame.Nombre
display(nombres)
type(nombres)
```

```
0      Marisa
1        Yudi
2    Carmilla
Name: Nombre, dtype: object
```

[82]:
```
pandas.core.series.Series
```

[83]:
```
#Acceso al primer nombre del DataFrame de frame

print(frame['Nombre'][0])
print(frame.Nombre[0])
print(nombres[0])
```

```
Marisa
Marisa
Marisa
```

Formas de Crear un DataFrame

1. Con una Serie de Pandas
2. Lista de diccionarios
3. Diccionario de Series en Pandas
4. Array de Numpy 2D
5. Array estructurado de Numpy

1.5.1 Modificar DataFrames

[84]:
```
#Añadir Columnas
diccionariox = {
    "Nombre": ["Alejandro" , "Martín", "PatricioCulon"],
    "Edad" : [34,29,12]
}

frame = pd.DataFrame(diccionariox, columns =␣
 ↪["Nacionalidad","Nombre","Edad","Profesion","Direccion"])
frame['Direccion'] = 'Desconocida'
display(frame)
```

	Nacionalidad	Nombre	Edad	Profesion	Direccion
0	NaN	Alejandro	34	NaN	Desconocida
1	NaN	Martín	29	NaN	Desconocida
2	NaN	PatricioCulon	12	NaN	Desconocida

[85]:
```
lista_direcciones = ['Rue 13 del Percebe, 13', 'Evergreen Terrace, 3', 'Av de␣
↪los Rombos, 12']
```

[86]:
```
frame['Direccion'] = lista_direcciones
display(frame)
```

```
  Nacionalidad         Nombre  Edad Profesion               Direccion
0          NaN      Alejandro    34       NaN  Rue 13 del Percebe, 13
1          NaN         Martín    29       NaN    Evergreen Terrace, 3
2          NaN  PatricioCulon    12       NaN    Av de los Rombos, 12
```

[87]:
```
#Añadir fila (requiere de todos los valores)

user_2 = ['Alemania', 'Klaus', 20, 'none', 'Desconocida']
frame.loc[3] = user_2
display(frame)
```

```
  Nacionalidad         Nombre  Edad Profesion               Direccion
0          NaN      Alejandro    34       NaN  Rue 13 del Percebe, 13
1          NaN         Martín    29       NaN    Evergreen Terrace, 3
2          NaN  PatricioCulon    12       NaN    Av de los Rombos, 12
3     Alemania          Klaus    20      none             Desconocida
```

[90]:
```
#Eliminar Fila

#Eliminar por índice
frame = pd.DataFrame(diccionario,␣
↪columns=['Nacionalidad','Nombre','Edad','Profesion'])
frame = frame.drop(2)
display(frame)

#Eliminar la columna Nombre
frame.drop('Nombre', axis = 1, inplace = True)
display(frame)
```

```
  Nacionalidad  Nombre  Edad Profesion
0          NaN  Marisa    34       NaN
1          NaN    Yudi    29       NaN
```

```
  Nacionalidad  Edad Profesion
0          NaN    34       NaN
1          NaN    29       NaN
```

[91]:
```
#eliminar columna
del frame['Profesion']
display(frame)
```

```
  Nacionalidad  Edad
0          NaN    34
1          NaN    29
```

[93]:
```
# acceder a la traspuesta (como una matriz)
#Intercambia filas x columnas, y columnas x filas
#Las columnas pasan a ser filas y Los índices pasan a ser columnas
#En matemáticas, la traspuesta de una matriz intercambia filas por columnas
#Pandas aplica el mismo concepto a DataFrame
display(frame.T)
```

```
                0    1
Nacionalidad  NaN  NaN
Edad           34   29
```

1.6 Iteración

[95]:
```
frame = pd.DataFrame(diccionario, columns =␣
 ↪['Nacionalidad','Nombre','Edad','Profesion'])
display(frame)

#Cuando iteras directamente sobre un DataFrame
#No recorres filas
#Recorres los nombres de las columnas.
for a in frame:
    print(a)
    print(type(a))
```

```
  Nacionalidad    Nombre  Edad Profesion
0          NaN    Marisa    34       NaN
1          NaN      Yudi    29       NaN
2          NaN  Carmilla    12       NaN
```

```
Nacionalidad
<class 'str'>
Nombre
<class 'str'>
Edad
<class 'str'>
Profesion
<class 'str'>
```

[96]:
```
# iteracion sobre filas
for value in frame.values:
    print(value)
    print(type(value))
```

```
[nan 'Marisa' 34 nan]
<class 'numpy.ndarray'>
[nan 'Yudi' 29 nan]
<class 'numpy.ndarray'>
[nan 'Carmilla' 12 nan]
<class 'numpy.ndarray'>
```

[97]:
```
# iterar sobre filas y luego sobre cada valor
for values in frame.values:
    for value in values:
        print(value)
```

```
nan
Marisa
34
nan
nan
Yudi
29
nan
nan
Carmilla
12
nan
```

1.6.1 Indexación y slicing con DataFrames

[101]:
```
d1 = {'ciudad':'Valencia', 'temperatura':10, 'o2':1}
d2 = {'ciudad':'Barcelona', 'temperatura':8}
d3 = {'ciudad':'Valencia', 'temperatura':9}
d4 = {'ciudad':'Madrid', 'temperatura':10, 'humedad':80}
d5 = {'ciudad':'Sevilla', 'temperatura':15, 'humedad':50, 'co2':6}
d6 = {'ciudad':'Valencia', 'temperatura':10, 'humedad':90, 'co2':10}

ls_data = [d1,d2,d3,d4,d5,d6]
df_data = pd.DataFrame(ls_data, index = list("abcdef"))
display(df_data)
```

	ciudad	temperatura	o2	humedad	co2
a	Valencia	10	1.0	NaN	NaN
b	Barcelona	8	NaN	NaN	NaN
c	Valencia	9	NaN	NaN	NaN
d	Madrid	10	NaN	80.0	NaN
e	Sevilla	15	NaN	50.0	6.0
f	Valencia	10	NaN	90.0	10.0

[102]:
```
#Acceso a un valor concreto por el índice por posición
print(df_data.iloc[1,1]) #fila 1 x col 1
```

```
8
```

[103]:
```
#Acceso a todos los valores hasta un índice por enteros
#Filas = todas las filas desde la posición 0 hasta la 2
#Col = todas las columnas desde la posición 0 hasta la 3
display(df_data.iloc[:3, :4])
```

```
      ciudad  temperatura   o2  humedad
a   Valencia           10  1.0      NaN
b  Barcelona            8  NaN      NaN
c   Valencia            9  NaN      NaN
```

[106]:
```
# Acceso a datos de manera explícita, indice semantico (se incluyen)

#accede a la fila con índice a y columna de temperatura
#Devuelve un valor
display(df_data.loc['a', 'temperatura'])
print("-----------------------------------")
#Selecciona las filas de a hacia c
#Seleciiona las columnas desde el inicio hasta o2
display(df_data.loc[:'c', :'o2'])
print("-----------------------------------")
#Selecciona todas las filas hasta c
#Selecciona las columnas desde temperatura hasta o2
display(df_data.loc[:'c', 'temperatura':'o2'])
print("-----------------------------------")
#Selecciona todas las filas
#Selecciona solo 2 columnas = ciudad y o2
display(df_data.loc[:, ['ciudad','o2']])
```

```
np.int64(10)
-----------------------------------
      ciudad  temperatura   o2
a   Valencia           10  1.0
b  Barcelona            8  NaN
c   Valencia            9  NaN
-----------------------------------
   temperatura   o2
a           10  1.0
b            8  NaN
c            9  NaN
-----------------------------------
      ciudad   o2
a   Valencia  1.0
b  Barcelona  NaN
c   Valencia  NaN
d     Madrid  NaN
e    Sevilla  NaN
f   Valencia  NaN
```

[108]:
```
# indexación con nombre de columna (por columnas)
```

```
#Aquí usas indexación por corchetes con el nombre de la columna.
#Al pasar un string, Pandas devuelve una Serie.
print(df_data['ciudad'])
#Aquí nos devuelve un dataframe
display(df_data[['ciudad', 'o2']])
```

```
a     Valencia
b    Barcelona
c     Valencia
d       Madrid
e      Sevilla
f     Valencia
Name: ciudad, dtype: object
```

	ciudad	o2
a	Valencia	1.0
b	Barcelona	NaN
c	Valencia	NaN
d	Madrid	NaN
e	Sevilla	NaN
f	Valencia	NaN

[109]:
```
# indexación con índice posicional (no permitido!).
#Pandas siempre interpreta algo como nombre de columna,
#no como índice de fila.
df_data[0] #saldrá error
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File ~\anaconda3\envs\jupyter_pdf\Lib\site-packages\pandas\core\indexes\base.py
 ↪3812, in Index.get_loc(self, key)
   3811 try:
-> 3812     return self._engine.get_loc(casted_key)
   3813 except KeyError as err:

File pandas/_libs/index.pyx:167, in pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/index.pyx:196, in pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/hashtable_class_helper.pxi:7088, in pandas._libs.hashtable.
 ↪PyObjectHashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:7096, in pandas._libs.hashtable.
 ↪PyObjectHashTable.get_item()

KeyError: 0

The above exception was the direct cause of the following exception:
```

```
KeyError                                  Traceback (most recent call last)
Cell In[109], line 3
      1 # indexación con índice posicional (no permitido!).
      2 #Esto busca columna.
----> 3 df_data[0]

File ~\anaconda3\envs\jupyter_pdf\Lib\site-packages\pandas\core\frame.py:4113,␣
 ↪in DataFrame.__getitem__(self, key)
   4111 if self.columns.nlevels > 1:
   4112     return self._getitem_multilevel(key)
-> 4113 indexer = self.columns.get_loc(key)
   4114 if is_integer(indexer):
   4115     indexer = [indexer]

File ~\anaconda3\envs\jupyter_pdf\Lib\site-packages\pandas\core\indexes\base.py
 ↪3819, in Index.get_loc(self, key)
   3814     if isinstance(casted_key, slice) or (
   3815         isinstance(casted_key, abc.Iterable)
   3816         and any(isinstance(x, slice) for x in casted_key)
   3817     ):
   3818         raise InvalidIndexError(key)
-> 3819     raise KeyError(key) from err
   3820 except TypeError:
   3821     # If we have a listlike key, _check_indexing_error will raise
   3822     #  InvalidIndexError. Otherwise we fall through and re-raise
   3823     #  the TypeError.
   3824     self._check_indexing_error(key)

KeyError: 0
```

[111]:
```
# indexar por posición con 'iloc'

#Series de la primera fila (qué marca los índices)
print(df_data.iloc[0]) #Devuelve la primera fila del DataFrame.
```

```
ciudad         Valencia
temperatura          10
o2                  1.0
humedad             NaN
co2                 NaN
Name: a, dtype: object
```

[113]:
```
# indexar semántico con 'loc'
# lo mismo de arriba pero para letras
df_data.loc['a'] # --> Series de la fila con índice 'a'
```

[113]:
```
ciudad         Valencia
temperatura          10
o2                  1.0
humedad             NaN
co2                 NaN
Name: a, dtype: object
```

[114]:
```
# indexar semántico con 'loc'
#Podemos buscar en un rango de índices, todo lo anterior hacia b
df_data.loc[:'b'] #a y b
```

[114]:

	ciudad	temperatura	o2	humedad	co2
a	Valencia	10	1.0	NaN	NaN
b	Barcelona	8	NaN	NaN	NaN

[117]:
```
# slicing anidado

#Busca las filas que sean anteriores hacia b (incluyendo b)
#en el otro loc, mostrar todas las filas anteriores pero solo mostrar las 2␣
 ↪columnas o2 y humedad
df_data.loc[:'b'].loc[:,["o2", "humedad"]]
```

[117]:

	o2	humedad
a	1.0	NaN
b	NaN	NaN

[124]:
```
#Si se modifica una porción del dataframe, se modifica el dataframe original
#Referencia

display(df_data)
print("----------------------------------------")

serie = df_data.loc['a']
print(serie)
print("----------------------------------------")

serie.iloc[2] = 3000
display(df_data)
print("----------------------------------------")
df_2 = df_data.loc('a').copy()
df_2.iloc[2]== 3000

display(df_2)
display(df_data)
```

	ciudad	temperatura	o2	humedad	co2
a	Valencia	10	1.0	NaN	NaN
b	Barcelona	8	NaN	NaN	NaN

```
c   Valencia            9  NaN      NaN   NaN
d     Madrid           10  NaN     80.0   NaN
e    Sevilla           15  NaN     50.0   6.0
f   Valencia           10  NaN     90.0  10.0

----------------------------------------
ciudad         Valencia
temperatura          10
o2                  1.0
humedad             NaN
co2                 NaN
Name: a, dtype: object
----------------------------------------

C:\Users\Usuario-PC\AppData\Local\Temp\ipykernel_28472\3619269315.py:11:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  serie.iloc[2] = 3000

      ciudad  temperatura   o2  humedad   co2
a   Valencia           10  1.0      NaN   NaN
b  Barcelona            8  NaN      NaN   NaN
c   Valencia            9  NaN      NaN   NaN
d     Madrid           10  NaN     80.0   NaN
e    Sevilla           15  NaN     50.0   6.0
f   Valencia           10  NaN     90.0  10.0

----------------------------------------
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\anaconda3\envs\jupyter_pdf\Lib\site-packages\pandas\core\generic.py in ?(cls,
 ↪axis)
    579             return cls._AXIS_TO_AXIS_NUMBER[axis]
    580         except KeyError:
--> 581             raise ValueError(f"No axis named {axis} for object type {cls.
 ↪__name__}")

KeyError: 'a'

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_28472\3619269315.py in ?()
     10
     11 serie.iloc[2] = 3000
```

```
     12 display(df_data)
     13 print("----------------------------------------")
---> 14 df_2 = df_data.loc('a').copy()
     15 df_2.iloc[2]== 3000
     16 display(df_2)
     17 display(df_data)

~\anaconda3\envs\jupyter_pdf\Lib\site-packages\pandas\core\indexing.py in ?
 ↪(self, axis)
    735         # we need to return a copy of ourselves
    736         new_self = type(self)(self.name, self.obj)
    737
    738         if axis is not None:
--> 739             axis_int_none = self.obj._get_axis_number(axis)
    740         else:
    741             axis_int_none = axis
    742         new_self.axis = axis_int_none

~\anaconda3\envs\jupyter_pdf\Lib\site-packages\pandas\core\generic.py in ?(cls,
 ↪axis)
    577     def _get_axis_number(cls, axis: Axis) -> AxisInt:
    578         try:
    579             return cls._AXIS_TO_AXIS_NUMBER[axis]
    580         except KeyError:
--> 581             raise ValueError(f"No axis named {axis} for object type {cl
 ↪__name__}")

ValueError: No axis named a for object type DataFrame
```

[125]:
```
df_data.loc(axis=1)['ciudad'] # --> equivalente frame['ciudad']
```

[125]:
```
a     Valencia
b    Barcelona
c     Valencia
d       Madrid
e      Sevilla
f     Valencia
Name: ciudad, dtype: object
```

[126]:
```
# qué problema puede tener este fragmento?
frame = pd.DataFrame({"Name" : ['Carlos','Pedro'], "Age" : [34,22]},
 ↪index=[1,0])
display(frame)
```

	Name	Age
1	Carlos	34
0	Pedro	22

[127]:
```
# por defecto, pandas interpreta índice posicional --> error en frames
# cuando hay posible ambigüedad, utilizar loc y iloc
print('Primera fila\n')
print(frame.iloc[0])
print('\nElemento con index 0\n')
print(frame.loc[0])
```

```
Primera fila

Name    Carlos
Age         34
Name: 1, dtype: object

Elemento con index 0

Name    Pedro
Age        22
Name: 0, dtype: object
```

1.6.2 Objeto Index de Pandas

[129]:
```
#Construcción de Índices

ind = pd.Index([2,3,5,23,26])

#recuperar datos
print(ind[3])
print(ind[::2])
```

```
23
Index([2, 5, 26], dtype='int64')
```

[132]:
```
#Usar un objeto index al crear dataframe
frame = pd.DataFrame({"Name" : ['Carlos','Pedro', 'Manolo', 'Luis', 'Alberto'],␣
 ↪"Age":[34,22,15,55,23]}, index = ind)

display(frame)
```

```
       Name  Age
2    Carlos   34
3     Pedro   22
5    Manolo   15
23     Luis   55
26  Alberto   23
```

[133]:
```
#Son inmutables, no se modifican los datos
ind[3] = 666
```

```
---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)
Cell In[133], line 2
      1 #Son inmutables, no se modifican los datos
----> 2 ind[3] = 666

File ~\anaconda3\envs\jupyter_pdf\Lib\site-packages\pandas\core\indexes\base.py
 ↪5383, in Index.__setitem__(self, key, value)
   5381 @final
   5382 def __setitem__(self, key, value) -> None:
-> 5383     raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

[137]:
```
#Cambiar índice de la columna
frame = pd.DataFrame({"Name" : ['Carlos','Pedro', 'Manolo', 'Luis', 'Alberto'],␣
 ↪"Age":[34,22,15,55,23]}, index = ind)
display(frame)
print("--------------------------------------------------")
#Usa la columna 'Age' como nuevo índice
#Elimina 'Age' de las columnas
#inplace=True modifica el DataFrame original
frame.set_index('Age', inplace= True)
display(frame)
```

```
       Name  Age
2    Carlos   34
3     Pedro   22
5    Manolo   15
23     Luis   55
26  Alberto   23

--------------------------------------------------

        Name
Age
34    Carlos
22     Pedro
15    Manolo
55      Luis
23   Alberto
```

1.6.3 Slicing

[140]:
```
d_and_d_characters = {
    'Name': ['bundenth','theorin','barlok'],
    'Strength': [10, 12, 19],
    'Wisdom': [20, 13, 6]
}
```

```
character_data = pd.DataFrame(d_and_d_characters, index=['a','b','c'])
print("-----------------------------------------")
display(character_data)
print("-----------------------------------------")
display(character_data[:-1])
print("-----------------------------------------")
display(character_data[1:2])
print("-----------------------------------------")
```

```
-----------------------------------------
```

	Name	Strength	Wisdom
a	bundenth	10	20
b	theorin	12	13
c	barlok	19	6

```
-----------------------------------------
```

	Name	Strength	Wisdom
a	bundenth	10	20
b	theorin	12	13

```
-----------------------------------------
```

	Name	Strength	Wisdom
b	theorin	12	13

```
-----------------------------------------
```

[141]:
```
# slicing para columnas
display(character_data[['Name','Wisdom']])
```

	Name	Wisdom
a	bundenth	20
b	theorin	13
c	barlok	6

[142]:
```
#slicing con 'loc' e 'iloc'

#.iloc trabaja solo con posiciones numéricas.
#Solo acepta números
#No incluye extremos
display(character_data.iloc[1:])
#.loc usa labels (nombres), no posiciones.
#incluye extremos
#Es semántico
display(character_data.loc[:'b','Name':'Strength'])
```

	Name	Strength	Wisdom
b	theorin	12	13
c	barlok	19	6

```
       Name  Strength
a  bundenth        10
b   theorin        12
```

¿Cómo filtrar filas y columnas? Por ejemplo, para todos los personajes, obtener 'Name' y 'Strength'

[143]:
```
#1) usando 'loc' para hacer slicing
display(character_data.loc[:,'Name':'Strength'])
```

```
       Name  Strength
a  bundenth        10
b   theorin        12
c    barlok        19
```

[144]:
```
#2) usando 'loc' para buscar específicamente filas y columnas
display(character_data.loc[ ['a','c'], ['Name','Wisdom'] ])
```

```
       Name  Wisdom
a  bundenth      20
c    barlok       6
```

[145]:
```
#3) Usando lo mismo pero esta vez con 'iloc'
#de la fila 0 hasta la 1, desde la columna 0 hasta la 1
display(character_data.iloc[[0,2],[0,2]])
#de la primera fila a la última, de la primera columna hasta la última
display(character_data.iloc[[0,-1],[0,-1]])
```

```
       Name  Wisdom
a  bundenth      20
c    barlok       6
```

```
       Name  Wisdom
a  bundenth      20
c    barlok       6
```

[146]:
```
# lista de los personajes con el atributo Strength > 11
display(character_data.loc[character_data['Strength'] > 11,␣
 ↪['Name','Strength']])
```

```
      Name  Strength
b  theorin        12
c   barlok        19
```

[147]:
```
 # listar los personajes con Strength > 15 o Wisdom > 15
display(character_data.loc[(character_data['Strength'] > 15) |␣
 ↪(character_data['Wisdom'] > 15)])
```

```
       Name  Strength  Wisdom
a  bundenth        10      20
c    barlok        19       6
```

1.7 Cargar y Guardar Datos en Pandas

[149]:
```
#Guardar a CSV
import os

ruta_archivo = os.path.join("res", "o_d_d_characters.csv")
#character_data.to_csv(ruta, sep=';') # sep por defecto: ','

loaded = pd.read_csv(ruta_archivo, sep=';')
display(loaded)
```

```
  Unnamed: 0      Name  Strength  Wisdom
0          a  bundenth        10      20
1          b   theorin        12      13
2          c    barlok        19       6
```

[150]:
```
ruta = os.path.join("res","titanic.csv")

titanic = pd.read_csv(ruta, sep=',')
display(titanic)

loaded = pd.read_csv(ruta, sep=',', index_col = 0)
display(loaded)
```

```
     PassengerId  Survived  Pclass  \
0              1         0       3
1              2         1       1
2              3         1       3
3              4         1       1
4              5         0       3
..           …       …     …
886          887         0       2
887          888         1       1
888          889         0       3
889          890         1       1
890          891         0       3

                                                  Name     Sex   Age  SibSp  \
0                              Braund, Mr. Owen Harris    male  22.0      1
1    Cumings, Mrs. John Bradley (Florence Briggs Th…  female  38.0      1
2                                 Heikkinen, Miss. Laina  female  26.0      0
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4                              Allen, Mr. William Henry    male  35.0      0
..                                                  …     …   …    …
886                              Montvila, Rev. Juozas    male  27.0      0
887                       Graham, Miss. Margaret Edith  female  19.0      0
888           Johnston, Miss. Catherine Helen "Carrie"  female   NaN      1
889                              Behr, Mr. Karl Howell    male  26.0      0
890                                Dooley, Mr. Patrick    male  32.0      0
```

```
     Parch            Ticket     Fare Cabin Embarked
0        0         A/5 21171   7.2500   NaN        S
1        0          PC 17599  71.2833   C85        C
2        0  STON/O2. 3101282   7.9250   NaN        S
3        0            113803  53.1000  C123        S
4        0            373450   8.0500   NaN        S
..     …                …       …   …       …
886      0            211536  13.0000   NaN        S
887      0            112053  30.0000   B42        S
888      2        W./C. 6607  23.4500   NaN        S
889      0            111369  30.0000  C148        C
890      0            370376   7.7500   NaN        Q

[891 rows x 12 columns]
             Survived  Pclass  \
PassengerId
1                   0       3
2                   1       1
3                   1       3
4                   1       1
5                   0       3
…               …     …
887                 0       2
888                 1       1
889                 0       3
890                 1       1
891                 0       3

                                                 Name     Sex   Age  \
PassengerId
1                                   Braund, Mr. Owen Harris    male  22.0
2           Cumings, Mrs. John Bradley (Florence Briggs Th…  female  38.0
3                                    Heikkinen, Miss. Laina  female  26.0
4             Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0
5                                  Allen, Mr. William Henry    male  35.0
…                                                       …     …   …
887                                  Montvila, Rev. Juozas    male  27.0
888                           Graham, Miss. Margaret Edith  female  19.0
889                Johnston, Miss. Catherine Helen "Carrie"  female   NaN
890                                    Behr, Mr. Karl Howell    male  26.0
891                                      Dooley, Mr. Patrick    male  32.0

             SibSp  Parch            Ticket     Fare Cabin Embarked
PassengerId
1                1      0         A/5 21171   7.2500   NaN        S
2                1      0          PC 17599  71.2833   C85        C
```

```
3                0      0  STON/O2. 3101282   7.9250   NaN        S
4                1      0            113803  53.1000  C123        S
5                0      0            373450   8.0500   NaN        S
…              …    …                …       …    …       …
887              0      0            211536  13.0000   NaN        S
888              0      0            112053  30.0000   B42        S
889              1      2        W./C. 6607  23.4500   NaN        S
890              0      0            111369  30.0000  C148        C
891              0      0            370376   7.7500   NaN        Q

[891 rows x 11 columns]
```

2 Ejercicio Práctico con Pandas

- MovieLens Dataset
- Review de Películas
- 1 millón de entradas
- Datos demográficos

[151]:
```
import numpy as np
import pandas as pd
#Descomponer archivos
import zipfile
#Para descargar URL
import urllib.request
import os

#1) Descargar MovieLens dataset

url = 'http://files.grouplens.org/datasets/movielens/ml-1m.zip'
ruta = os.path.join("res", "ml-1m.zip")
urllib.request.urlretrieve(url, ruta)
```

[151]:
```
('res\\ml-1m.zip', <http.client.HTTPMessage at 0x2056c0c5550>)
```

[152]:
```
#2) Descomponer archivo zip
ruta_ext = os.path.join("res")
with zipfile.ZipFile(ruta,'r') as z:
    print("Extrañendo todos los archivos...")
    z.extractall(ruta_ext)#este es el destino
    print("Ahora tenemos datos")
```

```
Extrañendo todos los archivos…
Ahora tenemos datos
```

[155]:
```
#3) Tomamos el README y revisamos los formatos
#Tendrá varios problemas:
#1. No tiene cabecera, el primer valor se pierde
```

```
#2. Las columnas no tienen nombres

ruta_users = os.path.join("res", "ml-1m", "users.dat")
users_dataset = pd.read_csv(ruta_users, sep='::', index_col=0, engine='python')
display(users_dataset)
```

```
      F  1.1  10  48067
1
2     M   56  16  70072
3     M   25  15  55117
4     M   45   7  02460
5     M   25  20  55455
6     F   50   9  55117
…   ..  …   ..    …
6036  F   25  15  32603
6037  F   45   1  76006
6038  F   56   1  14706
6039  F   45   0  01060
6040  M   25   6  11106

[6039 rows x 4 columns]
```

[156]:
```
pd.read_csv?
```

```
Signature:
pd.read_csv(
    filepath_or_buffer: 'FilePath | ReadCsvBuffer[bytes] | ReadCsvBuffer[str]',
    *,
    sep: 'str | None | lib.NoDefault' = <no_default>,
    delimiter: 'str | None | lib.NoDefault' = None,
    header: "int | Sequence[int] | None | Literal['infer']" = 'infer',
    names: 'Sequence[Hashable] | None | lib.NoDefault' = <no_default>,
    index_col: 'IndexLabel | Literal[False] | None' = None,
    usecols: 'UsecolsArgType' = None,
    dtype: 'DtypeArg | None' = None,
    engine: 'CSVEngine | None' = None,
    converters: 'Mapping[Hashable, Callable] | None' = None,
    true_values: 'list | None' = None,
    false_values: 'list | None' = None,
    skipinitialspace: 'bool' = False,
    skiprows: 'list[int] | int | Callable[[Hashable], bool] | None' = None,
    skipfooter: 'int' = 0,
    nrows: 'int | None' = None,
    na_values: 'Hashable | Iterable[Hashable] | Mapping[Hashable,␣
 ↪Iterable[Hashable]] | None' = None,
    keep_default_na: 'bool' = True,
    na_filter: 'bool' = True,
    verbose: 'bool | lib.NoDefault' = <no_default>,
```

```
    skip_blank_lines: 'bool' = True,
    parse_dates: 'bool | Sequence[Hashable] | None' = None,
    infer_datetime_format: 'bool | lib.NoDefault' = <no_default>,
    keep_date_col: 'bool | lib.NoDefault' = <no_default>,
    date_parser: 'Callable | lib.NoDefault' = <no_default>,
    date_format: 'str | dict[Hashable, str] | None' = None,
    dayfirst: 'bool' = False,
    cache_dates: 'bool' = True,
    iterator: 'bool' = False,
    chunksize: 'int | None' = None,
    compression: 'CompressionOptions' = 'infer',
    thousands: 'str | None' = None,
    decimal: 'str' = '.',
    lineterminator: 'str | None' = None,
    quotechar: 'str' = '"',
    quoting: 'int' = 0,
    doublequote: 'bool' = True,
    escapechar: 'str | None' = None,
    comment: 'str | None' = None,
    encoding: 'str | None' = None,
    encoding_errors: 'str | None' = 'strict',
    dialect: 'str | csv.Dialect | None' = None,
    on_bad_lines: 'str' = 'error',
    delim_whitespace: 'bool | lib.NoDefault' = <no_default>,
    low_memory: 'bool' = True,
    memory_map: 'bool' = False,
    float_precision: "Literal['high', 'legacy'] | None" = None,
    storage_options: 'StorageOptions | None' = None,
    dtype_backend: 'DtypeBackend | lib.NoDefault' = <no_default>,
) -> 'DataFrame | TextFileReader'
Docstring:
Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file
into chunks.

Additional help can be found in the online docs for
`IO Tools <https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html>`_.

Parameters
----------
filepath_or_buffer : str, path object or file-like object
    Any valid string path is acceptable. The string could be a URL. Valid
    URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is
    expected. A local file could be: file://localhost/path/to/table.csv.

    If you want to pass in a path object, pandas accepts any ``os.PathLike``.
```

```
    By file-like object, we refer to objects with a ``read()`` method, such as
    a file handle (e.g. via builtin ``open`` function) or ``StringIO``.
sep : str, default ','
    Character or regex pattern to treat as the delimiter. If ``sep=None``, the
    C engine cannot automatically detect
    the separator, but the Python parsing engine can, meaning the latter will
    be used and automatically detect the separator from only the first valid
    row of the file by Python's builtin sniffer tool, ``csv.Sniffer``.
    In addition, separators longer than 1 character and different from
    ``'\s+'`` will be interpreted as regular expressions and will also force
    the use of the Python parsing engine. Note that regex delimiters are prone
    to ignoring quoted data. Regex example: ``'\r\t'``.
delimiter : str, optional
    Alias for ``sep``.
header : int, Sequence of int, 'infer' or None, default 'infer'
    Row number(s) containing column labels and marking the start of the
    data (zero-indexed). Default behavior is to infer the column names: if no␣
 ↪``names``
    are passed the behavior is identical to ``header=0`` and column
    names are inferred from the first line of the file, if column
    names are passed explicitly to ``names`` then the behavior is identical to
    ``header=None``. Explicitly pass ``header=0`` to be able to
    replace existing names. The header can be a list of integers that
    specify row locations for a :class:`~pandas.MultiIndex` on the columns
    e.g. ``[0, 1, 3]``. Intervening rows that are not specified will be
    skipped (e.g. 2 in this example is skipped). Note that this
    parameter ignores commented lines and empty lines if
    ``skip_blank_lines=True``, so ``header=0`` denotes the first line of
    data rather than the first line of the file.
names : Sequence of Hashable, optional
    Sequence of column labels to apply. If the file contains a header row,
    then you should explicitly pass ``header=0`` to override the column names.
    Duplicates in this list are not allowed.
index_col : Hashable, Sequence of Hashable or False, optional
  Column(s) to use as row label(s), denoted either by column labels or column
  indices.  If a sequence of labels or indices is given, :class:`~pandas.
 ↪MultiIndex`
  will be formed for the row labels.

  Note: ``index_col=False`` can be used to force pandas to *not* use the first
  column as the index, e.g., when you have a malformed file with delimiters at
  the end of each line.
usecols : Sequence of Hashable or Callable, optional
    Subset of columns to select, denoted either by column labels or column␣
 ↪indices.
    If list-like, all elements must either
    be positional (i.e. integer indices into the document columns) or strings
```

```
    that correspond to column names provided either by the user in ``names`` or
    inferred from the document header row(s). If ``names`` are given, the␣
 ↪document
    header row(s) are not taken into account. For example, a valid list-like
    ``usecols`` parameter would be ``[0, 1, 2]`` or ``['foo', 'bar', 'baz']``.
    Element order is ignored, so ``usecols=[0, 1]`` is the same as ``[1, 0]``.
    To instantiate a :class:`~pandas.DataFrame` from ``data`` with element order
    preserved use ``pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]``
    for columns in ``['foo', 'bar']`` order or
    ``pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]``
    for ``['bar', 'foo']`` order.

    If callable, the callable function will be evaluated against the column
    names, returning names where the callable function evaluates to ``True``. An
    example of a valid callable argument would be ``lambda x: x.upper() in
    ['AAA', 'BBB', 'DDD']``. Using this parameter results in much faster
    parsing time and lower memory usage.
dtype : dtype or dict of {Hashable : dtype}, optional
    Data type(s) to apply to either the whole dataset or individual columns.
    E.g., ``{'a': np.float64, 'b': np.int32, 'c': 'Int64'}``
    Use ``str`` or ``object`` together with suitable ``na_values`` settings
    to preserve and not interpret ``dtype``.
    If ``converters`` are specified, they will be applied INSTEAD
    of ``dtype`` conversion.

    .. versionadded:: 1.5.0

        Support for ``defaultdict`` was added. Specify a ``defaultdict`` as␣
 ↪input where
        the default determines the ``dtype`` of the columns which are not␣
 ↪explicitly
        listed.
engine : {'c', 'python', 'pyarrow'}, optional
    Parser engine to use. The C and pyarrow engines are faster, while the python␣
 ↪engine
    is currently more feature-complete. Multithreading is currently only␣
 ↪supported by
    the pyarrow engine.

    .. versionadded:: 1.4.0

        The 'pyarrow' engine was added as an *experimental* engine, and some␣
 ↪features
        are unsupported, or may not work correctly, with this engine.
converters : dict of {Hashable : Callable}, optional
    Functions for converting values in specified columns. Keys can either
    be column labels or column indices.
```

```
true_values : list, optional
    Values to consider as ``True`` in addition to case-insensitive variants of␣
 ↪'True'.
false_values : list, optional
    Values to consider as ``False`` in addition to case-insensitive variants of␣
 ↪'False'.
skipinitialspace : bool, default False
    Skip spaces after delimiter.
skiprows : int, list of int or Callable, optional
    Line numbers to skip (0-indexed) or number of lines to skip (``int``)
    at the start of the file.

    If callable, the callable function will be evaluated against the row
    indices, returning ``True`` if the row should be skipped and ``False``␣
 ↪otherwise.
    An example of a valid callable argument would be ``lambda x: x in [0, 2]``.
skipfooter : int, default 0
    Number of lines at bottom of file to skip (Unsupported with ``engine='c'``).
nrows : int, optional
    Number of rows of file to read. Useful for reading pieces of large files.
na_values : Hashable, Iterable of Hashable or dict of {Hashable : Iterable},␣
 ↪optional
    Additional strings to recognize as ``NA``/``NaN``. If ``dict`` passed,␣
 ↪specific
    per-column ``NA`` values.  By default the following values are interpreted as
    ``NaN``: " ", "#N/A", "#N/A N/A", "#NA", "-1.#IND", "-1.#QNAN", "-NaN",␣
 ↪"-nan",
    "1.#IND", "1.#QNAN", "<NA>", "N/A", "NA", "NULL", "NaN", "None",
    "n/a", "nan", "null ".

keep_default_na : bool, default True
    Whether or not to include the default ``NaN`` values when parsing the data.
    Depending on whether ``na_values`` is passed in, the behavior is as follows:

    * If ``keep_default_na`` is ``True``, and ``na_values`` are specified,␣
 ↪``na_values``
      is appended to the default ``NaN`` values used for parsing.
    * If ``keep_default_na`` is ``True``, and ``na_values`` are not specified,␣
 ↪only
      the default ``NaN`` values are used for parsing.
    * If ``keep_default_na`` is ``False``, and ``na_values`` are specified, only
      the ``NaN`` values specified ``na_values`` are used for parsing.
    * If ``keep_default_na`` is ``False``, and ``na_values`` are not specified,␣
 ↪no
      strings will be parsed as ``NaN``.
```

```
    Note that if ``na_filter`` is passed in as ``False``, the␣
 ↪``keep_default_na`` and
    ``na_values`` parameters will be ignored.
na_filter : bool, default True
    Detect missing value markers (empty strings and the value of ``na_values``).␣
 ↪In
    data without any ``NA`` values, passing ``na_filter=False`` can improve the
    performance of reading a large file.
verbose : bool, default False
    Indicate number of ``NA`` values placed in non-numeric columns.

    .. deprecated:: 2.2.0
skip_blank_lines : bool, default True
    If ``True``, skip over blank lines rather than interpreting as ``NaN``␣
 ↪values.
parse_dates : bool, list of Hashable, list of lists or dict of {Hashable :␣
 ↪list}, default False
    The behavior is as follows:

    * ``bool``. If ``True`` -> try parsing the index. Note: Automatically set to
      ``True`` if ``date_format`` or ``date_parser`` arguments have been passed.
    * ``list`` of ``int`` or names. e.g. If ``[1, 2, 3]`` -> try parsing columns␣
 ↪1, 2, 3
      each as a separate date column.
    * ``list`` of ``list``. e.g.  If ``[[1, 3]]`` -> combine columns 1 and 3 and␣
 ↪parse
      as a single date column. Values are joined with a space before parsing.
    * ``dict``, e.g. ``{'foo' : [1, 3]}`` -> parse columns 1, 3 as date and call
      result 'foo'. Values are joined with a space before parsing.

    If a column or index cannot be represented as an array of ``datetime``,
    say because of an unparsable value or a mixture of timezones, the column
    or index will be returned unaltered as an ``object`` data type. For
    non-standard ``datetime`` parsing, use :func:`~pandas.to_datetime` after
    :func:`~pandas.read_csv`.

    Note: A fast-path exists for iso8601-formatted dates.
infer_datetime_format : bool, default False
    If ``True`` and ``parse_dates`` is enabled, pandas will attempt to infer the
    format of the ``datetime`` strings in the columns, and if it can be inferred,
    switch to a faster method of parsing them. In some cases this can increase
    the parsing speed by 5-10x.

    .. deprecated:: 2.0.0
        A strict version of this argument is now the default, passing it has no␣
 ↪effect.
```

```
keep_date_col : bool, default False
    If ``True`` and ``parse_dates`` specifies combining multiple columns then
    keep the original columns.
date_parser : Callable, optional
    Function to use for converting a sequence of string columns to an array of
    ``datetime`` instances. The default uses ``dateutil.parser.parser`` to do the
    conversion. pandas will try to call ``date_parser`` in three different ways,
    advancing to the next if an exception occurs: 1) Pass one or more arrays
    (as defined by ``parse_dates``) as arguments; 2) concatenate (row-wise) the
    string values from the columns defined by ``parse_dates`` into a single array
    and pass that; and 3) call ``date_parser`` once for each row using one or
    more strings (corresponding to the columns defined by ``parse_dates``) as
    arguments.

    .. deprecated:: 2.0.0
       Use ``date_format`` instead, or read in as ``object`` and then apply
       :func:`~pandas.to_datetime` as-needed.
date_format : str or dict of column -> format, optional
    Format to use for parsing dates when used in conjunction with␣
 ↪``parse_dates``.
    The strftime to parse time, e.g. :const:`"%d/%m/%Y"`. See
    `strftime documentation
    <https://docs.python.org/3/library/datetime.html
    #strftime-and-strptime-behavior>`_ for more information on choices, though
    note that :const:`"%f"` will parse all the way up to nanoseconds.
    You can also pass:

    - "ISO8601", to parse any `ISO8601 <https://en.wikipedia.org/wiki/ISO_8601>`_
        time string (not necessarily in exactly the same format);
    - "mixed", to infer the format for each element individually. This is risky,
        and you should probably use it along with `dayfirst`.

    .. versionadded:: 2.0.0
dayfirst : bool, default False
    DD/MM format dates, international and European format.
cache_dates : bool, default True
    If ``True``, use a cache of unique, converted dates to apply the ``datetime``
    conversion. May produce significant speed-up when parsing duplicate
    date strings, especially ones with timezone offsets.

iterator : bool, default False
    Return ``TextFileReader`` object for iteration or getting chunks with
    ``get_chunk()``.
chunksize : int, optional
    Number of lines to read from the file per chunk. Passing a value will cause␣
 ↪the
    function to return a ``TextFileReader`` object for iteration.
```

```
    See the `IO Tools docs
    <https://pandas.pydata.org/pandas-docs/stable/io.html#io-chunking>`_
    for more information on ``iterator`` and ``chunksize``.

compression : str or dict, default 'infer'
    For on-the-fly decompression of on-disk data. If 'infer' and␣
 ↪'filepath_or_buffer' is
    path-like, then detect compression from the following extensions: '.gz',
    '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2'
    (otherwise no compression).
    If using 'zip' or 'tar', the ZIP file must contain only one data file to be␣
 ↪read in.
    Set to ``None`` for no decompression.
    Can also be a dict with key ``'method'`` set
    to one of {``'zip'``, ``'gzip'``, ``'bz2'``, ``'zstd'``, ``'xz'``,␣
 ↪``'tar'``} and
    other key-value pairs are forwarded to
    ``zipfile.ZipFile``, ``gzip.GzipFile``,
    ``bz2.BZ2File``, ``zstandard.ZstdDecompressor``, ``lzma.LZMAFile`` or
    ``tarfile.TarFile``, respectively.
    As an example, the following could be passed for Zstandard decompression␣
 ↪using a
    custom compression dictionary:
    ``compression={'method': 'zstd', 'dict_data': my_compression_dict}``.

    .. versionadded:: 1.5.0
        Added support for `.tar` files.

    .. versionchanged:: 1.4.0 Zstandard support.

thousands : str (length 1), optional
    Character acting as the thousands separator in numerical values.
decimal : str (length 1), default '.'
    Character to recognize as decimal point (e.g., use ',' for European data).
lineterminator : str (length 1), optional
    Character used to denote a line break. Only valid with C parser.
quotechar : str (length 1), optional
    Character used to denote the start and end of a quoted item. Quoted
    items can include the ``delimiter`` and it will be ignored.
quoting : {0 or csv.QUOTE_MINIMAL, 1 or csv.QUOTE_ALL, 2 or csv.
 ↪QUOTE_NONNUMERIC, 3 or csv.QUOTE_NONE}, default csv.QUOTE_MINIMAL
    Control field quoting behavior per ``csv.QUOTE_*`` constants. Default is
    ``csv.QUOTE_MINIMAL`` (i.e., 0) which implies that only fields containing␣
 ↪special
    characters are quoted (e.g., characters defined in ``quotechar``,␣
 ↪``delimiter``,
    or ``lineterminator``.
```

```
doublequote : bool, default True
   When ``quotechar`` is specified and ``quoting`` is not ``QUOTE_NONE``,␣
 ↪indicate
   whether or not to interpret two consecutive ``quotechar`` elements INSIDE a
   field as a single ``quotechar`` element.
escapechar : str (length 1), optional
    Character used to escape other characters.
comment : str (length 1), optional
    Character indicating that the remainder of line should not be parsed.
    If found at the beginning
    of a line, the line will be ignored altogether. This parameter must be a
    single character. Like empty lines (as long as ``skip_blank_lines=True``),
    fully commented lines are ignored by the parameter ``header`` but not by
    ``skiprows``. For example, if ``comment='#'``, parsing
    ``#empty\na,b,c\n1,2,3`` with ``header=0`` will result in ``'a,b,c'`` being
    treated as the header.
encoding : str, optional, default 'utf-8'
    Encoding to use for UTF when reading/writing (ex. ``'utf-8'``). `List of␣
 ↪Python
    standard encodings
    <https://docs.python.org/3/library/codecs.html#standard-encodings>`_ .

encoding_errors : str, optional, default 'strict'
    How encoding errors are treated. `List of possible values
    <https://docs.python.org/3/library/codecs.html#error-handlers>`_ .

    .. versionadded:: 1.3.0

dialect : str or csv.Dialect, optional
    If provided, this parameter will override values (default or not) for the
    following parameters: ``delimiter``, ``doublequote``, ``escapechar``,
    ``skipinitialspace``, ``quotechar``, and ``quoting``. If it is necessary to
    override values, a ``ParserWarning`` will be issued. See ``csv.Dialect``
    documentation for more details.
on_bad_lines : {'error', 'warn', 'skip'} or Callable, default 'error'
    Specifies what to do upon encountering a bad line (a line with too many␣
 ↪fields).
    Allowed values are :

    - ``'error'``, raise an Exception when a bad line is encountered.
    - ``'warn'``, raise a warning when a bad line is encountered and skip that␣
 ↪line.
    - ``'skip'``, skip bad lines without raising or warning when they are␣
 ↪encountered.

    .. versionadded:: 1.3.0
```

```
    .. versionadded:: 1.4.0

        - Callable, function with signature
          ``(bad_line: list[str]) -> list[str] | None`` that will process a␣
↪single
          bad line. ``bad_line`` is a list of strings split by the ``sep``.
          If the function returns ``None``, the bad line will be ignored.
          If the function returns a new ``list`` of strings with more elements␣
↪than
          expected, a ``ParserWarning`` will be emitted while dropping extra␣
↪elements.
          Only supported when ``engine='python'``

    .. versionchanged:: 2.2.0

        - Callable, function with signature
          as described in `pyarrow documentation
          <https://arrow.apache.org/docs/python/generated/pyarrow.csv.
↪ParseOptions.html
          #pyarrow.csv.ParseOptions.invalid_row_handler>`_ when␣
↪``engine='pyarrow'``

delim_whitespace : bool, default False
    Specifies whether or not whitespace (e.g. ``' '`` or ``'\t'``) will be
    used as the ``sep`` delimiter. Equivalent to setting ``sep='\s+'``. If this␣
↪option
    is set to ``True``, nothing should be passed in for the ``delimiter``
    parameter.

    .. deprecated:: 2.2.0
        Use ``sep="\s+"`` instead.
low_memory : bool, default True
    Internally process the file in chunks, resulting in lower memory use
    while parsing, but possibly mixed type inference.  To ensure no mixed
    types either set ``False``, or specify the type with the ``dtype`` parameter.
    Note that the entire file is read into a single :class:`~pandas.DataFrame`
    regardless, use the ``chunksize`` or ``iterator`` parameter to return the␣
↪data in
    chunks. (Only valid with C parser).
memory_map : bool, default False
    If a filepath is provided for ``filepath_or_buffer``, map the file object
    directly onto memory and access the data directly from there. Using this
    option can improve performance because there is no longer any I/O overhead.
float_precision : {'high', 'legacy', 'round_trip'}, optional
    Specifies which converter the C engine should use for floating-point
    values. The options are ``None`` or ``'high'`` for the ordinary converter,
    ``'legacy'`` for the original lower precision pandas converter, and
```

```
    ``'round_trip'`` for the round-trip converter.

storage_options : dict, optional
    Extra options that make sense for a particular storage connection, e.g.
    host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
    are forwarded to ``urllib.request.Request`` as header options. For other
    URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are
    forwarded to ``fsspec.open``. Please see ``fsspec`` and ``urllib`` for more
    details, and for more examples on storage options refer `here
    <https://pandas.pydata.org/docs/user_guide/io.html?
    highlight=storage_options#reading-writing-remote-files>`_.

dtype_backend : {'numpy_nullable', 'pyarrow'}, default 'numpy_nullable'
    Back-end data type applied to the resultant :class:`DataFrame`
    (still experimental). Behaviour is as follows:

    * ``"numpy_nullable"``: returns nullable-dtype-backed :class:`DataFrame`
      (default).
    * ``"pyarrow"``: returns pyarrow-backed nullable :class:`ArrowDtype`
      DataFrame.

    .. versionadded:: 2.0

Returns
-------
DataFrame or TextFileReader
    A comma-separated values (csv) file is returned as two-dimensional
    data structure with labeled axes.

See Also
--------
DataFrame.to_csv : Write DataFrame to a comma-separated values (csv) file.
read_table : Read general delimited file into DataFrame.
read_fwf : Read a table of fixed-width formatted lines into DataFrame.

Examples
--------
>>> pd.read_csv('data.csv')  # doctest: +SKIP
File:      c:
 ↪\users\usuario-pc\anaconda3\envs\jupyter_pdf\lib\site-packages\pandas\io\parsers\readers.
 ↪py
Type:      function
```

[157]:
```
#Especificar nombres, cargar sin cabecera

users_dataset = pd.read_csv(ruta_users, sep='::', index_col=0, header=None,␣
 ↪names=['UserID','Gender','Age','Occupation','Zip-code'], engine='python')
```

```
display(users_dataset)
```

```
       Gender  Age  Occupation Zip-code
UserID
1           F    1          10    48067
2           M   56          16    70072
3           M   25          15    55117
4           M   45           7    02460
5           M   25          20    55455
…         …   …           …       …
6036        F   25          15    32603
6037        F   45           1    76006
6038        F   56           1    14706
6039        F   45           0    01060
6040        M   25           6    11106

[6040 rows x 4 columns]
```

[159]:
```
# samplear la tabla

#selecciona 10 filas aleatorias del DataFrame
display(users_dataset.sample(10))
```

```
       Gender  Age  Occupation Zip-code
UserID
5849        M   35          17    97124
5179        M   25          12    94110
2465        M   50           6    44333
4257        M   50          16    38122
2949        M   25          17    98033
3901        M   18          14    85282
3335        M   25           0    10023
2391        M   50          18    13126
3400        M   25          14    08742
2621        M   18           0    46304
```

[162]:
```
# samplear la cabeza

#.head(n) devuelve las primeras n filas del DataFrame
#en este caso, las 4 primeras filas
display(users_dataset.head(4))
```

```
       Gender  Age  Occupation Zip-code
UserID
1           F    1          10    48067
2           M   56          16    70072
3           M   25          15    55117
4           M   45           7    02460
```

[164]:
```
# samplear la cola

#.tail(n) devuelve las últimas n filas del DataFrame
#en este caso, las 10 últimas filas
display(users_dataset.tail(10))
```

```
       Gender  Age  Occupation Zip-code
UserID
6031        F   18           0    45123
6032        M   45           7    55108
6033        M   50          13    78232
6034        M   25          14    94117
6035        F   25           1    78734
6036        F   25          15    32603
6037        F   45           1    76006
6038        F   56           1    14706
6039        F   45           0    01060
6040        M   25           6    11106
```

[165]:
```
# tipos de datos sobre las columnas
users_dataset.dtypes
```

[165]:
```
Gender        object
Age            int64
Occupation     int64
Zip-code      object
dtype: object
```

[166]:
```
#Accede a la columna Zip-code del DataFrame.
#Devuelve una pd.Series
#Devuelve la longitud de cada string parseando a string
#Crea una máscara booleana:
#True si la longitud es mayor que 5
#False si la longitud es menor o igual a 5
#Devuelve solo las filas donde la condición es True
display(users_dataset[users_dataset['Zip-code'].str.len() > 5])
```

```
       Gender  Age  Occupation    Zip-code
UserID
161         M   45          16  98107-2117
233         F   45          20  37919-4204
293         M   56           1  55337-4056
458         M   50          16  55405-2546
506         M   25          16  55103-1006
…         …   …           …          …
5682        M   18           0  23455-4959
5904        F   45          12      954025
5925        F   25           0  90035-4444
```

```
5967        M   50          16  73069-5429
5985        F   18           4  78705-5221

[81 rows x 4 columns]
```

[167]:
```
# información general sobre atributos numéricos
display(users_dataset.describe())
```

```
                Age   Occupation
count  6040.000000  6040.000000
mean     30.639238     8.146854
std      12.895962     6.329511
min       1.000000     0.000000
25%      25.000000     3.000000
50%      25.000000     7.000000
75%      35.000000    14.000000
max      56.000000    20.000000
```

[168]:
```
users_dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 6040 entries, 1 to 6040
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Gender      6040 non-null   object
 1   Age         6040 non-null   int64
 2   Occupation  6040 non-null   int64
 3   Zip-code    6040 non-null   object
dtypes: int64(2), object(2)
memory usage: 235.9+ KB
```

[169]:
```
# incluir otros atributos (no todo tiene sentido)
display(users_dataset.describe(include='all'))
```

```
       Gender          Age   Occupation Zip-code
count    6040  6040.000000  6040.000000     6040
unique      2          NaN          NaN     3439
top         M          NaN          NaN    48104
freq     4331          NaN          NaN       19
mean      NaN    30.639238     8.146854      NaN
std       NaN    12.895962     6.329511      NaN
min       NaN     1.000000     0.000000      NaN
25%       NaN    25.000000     3.000000      NaN
50%       NaN    25.000000     7.000000      NaN
75%       NaN    35.000000    14.000000      NaN
max       NaN    56.000000    20.000000      NaN
```

[172]:
```
# cuántos usuarios son mujeres (Gender='F')
len(users_dataset[users_dataset['Gender'] == 'F'])
#así sería en una query: select count(*) from users_dataset where users_dataset.
 ↪Gender = 'F'
```

[172]: 1709

[176]:
```
# mostrar solo los menores de edad
under_age = users_dataset[users_dataset['Age'] == 1]
print(len(under_age))
display(under_age.sample(10))
```

222

UserID	Gender	Age	Occupation	Zip-code
926	M	1	10	07869
4541	M	1	10	55427
4087	M	1	4	63376
866	M	1	10	08820
871	M	1	10	76013
3537	M	1	10	97402
1327	M	1	10	12159
1365	F	1	10	61665
4478	M	1	12	02138
5228	M	1	0	75070

[178]:
```
# Leer el archivo de usuarios desde un CSV
# ruta_users: ruta del archivo
# sep='::' indica que el separador es '::'
# index_col=0 usa la primera columna como índice
# header=None indica que el archivo no tiene encabezado
# names define los nombres de las columnas
# engine='python' es necesario porque el separador tiene más de un carácter
users_dataset = pd.read_csv(
    ruta_users,
    sep='::',
    index_col=0,
    header=None,
    names=['UserID', 'Gender', 'Age', 'Occupation', 'Zip-code'],
    engine='python'
)

# Filtrar los usuarios cuya edad es igual a 1
# Esto crea un NUEVO DataFrame que puede ser una vista del original
# (no una copia segura)
under_age = users_dataset[users_dataset['Age'] == 1]
```

```
# Intentar modificar la columna 'Age' del DataFrame filtrado
# Se asigna NaN a la edad
# Esto provoca SettingWithCopyWarning porque 'under_age'
# puede ser solo una vista del DataFrame original
under_age.loc[:, 'Age'] = np.nan

# Mostrar las primeras filas del DataFrame resultante
display(under_age.head())

# users_dataset[users_dataset['Age'] < 18] = under_age
# display(users_dataset)
```

```
C:\Users\Usuario-PC\AppData\Local\Temp\ipykernel_28472\3723279683.py:26:
FutureWarning: Setting an item of incompatible dtype is deprecated and will
raise in a future error of pandas. Value 'nan' has dtype incompatible with
int64, please explicitly cast to a compatible dtype first.
  under_age.loc[:, 'Age'] = np.nan
```

UserID	Gender	Age	Occupation	Zip-code
1	F	NaN	10	48067
19	M	NaN	10	48073
51	F	NaN	10	10562
75	F	NaN	10	01748
86	F	NaN	10	54467

[180]:
```
# Leer el archivo CSV de usuarios
# - sep='::' = separador de dos caracteres
# - index_col=0 = la primera columna será el índice
# - header=None = el CSV no tiene encabezados
# - names = nombres manuales de las columnas
# - engine='python' = necesario para separadores largos
users_dataset = pd.read_csv(
    ruta_users,
    sep='::',
    index_col=0,
    header=None,
    names=['UserID', 'Gender', 'Age', 'Occupation', 'Zip-code'],
    engine='python'
)

# Filtrar los usuarios con edad incorrecta (edad == 1)
# Se obtiene un subconjunto del DataFrame original
under_age = users_dataset[users_dataset['Age'] == 1]

# Crear una COPIA explícita del subconjunto
# Esto evita el SettingWithCopyWarning
under_age_copy = under_age.copy()
```

```python
# Mostrar las primeras filas antes de modificar
display(under_age_copy.head())
print("-----------------------------------------------")

# Reemplazar la edad incorrecta por NaN
# Aquí ya no hay warning porque estamos trabajando sobre una copia real
under_age_copy['Age'] = np.nan

# Mostrar las filas después de la modificación
display(under_age_copy.head())
print("-----------------------------------------------")
# Reemplazar en el DataFrame original las filas cuyo índice coincide
# con las filas modificadas en under_age_copy
# Pandas alinea por índices automáticamente
users_dataset[users_dataset['Age'] == 1] = under_age_copy

# Mostrar las primeras filas del DataFrame final
display(users_dataset.head())
print("-----------------------------------------------")
```

```
       Gender  Age  Occupation Zip-code
UserID
1           F    1          10    48067
19          M    1          10    48073
51          F    1          10    10562
75          F    1          10    01748
86          F    1          10    54467

-----------------------------------------------

       Gender  Age  Occupation Zip-code
UserID
1           F  NaN          10    48067
19          M  NaN          10    48073
51          F  NaN          10    10562
75          F  NaN          10    01748
86          F  NaN          10    54467

-----------------------------------------------

       Gender   Age  Occupation Zip-code
UserID
1           F   NaN          10    48067
2           M  56.0          16    70072
3           M  25.0          15    55117
4           M  45.0           7    02460
5           M  25.0          20    55455

-----------------------------------------------
```

[182]:
```python
# Leer el archivo CSV que contiene el dataset de usuarios
# sep='::' indica que el separador del archivo es '::'
# index_col=0 indica que la primera columna será usada como índice
# header=None indica que el archivo no tiene fila de encabezados
# names asigna los nombres de las columnas manualmente
# engine='python' es obligatorio cuando el separador tiene más de un carácter
users_dataset = pd.read_csv(
    ruta_users,
    sep='::',
    index_col=0,
    header=None,
    names=['UserID', 'Gender', 'Age', 'Occupation', 'Zip-code'],
    engine='python'
)

# Mostrar las primeras 4 filas donde la edad es incorrecta (edad igual a 1)
# Esto permite verificar qué registros serán tratados
display(users_dataset[users_dataset['Age'] == 1].head(4))
print("-----------------------------------------------")
# Reemplazar las edades incorrectas (Age == 1) por NaN
# Se usa loc para modificar el DataFrame original de forma segura
users_dataset.loc[users_dataset['Age'] == 1, 'Age'] = np.nan

# Mostrar el DataFrame completo después de reemplazar las edades incorrectas
display(users_dataset)
print("-----------------------------------------------")
# Filtrar las filas que ahora tienen valores NaN en la columna Age
# pd.isnull detecta los valores faltantes
display(users_dataset.loc[pd.isnull(users_dataset['Age'])].head(4))
print("-----------------------------------------------")
# Eliminar del DataFrame todas las filas cuya edad es NaN
# Primero se obtienen los índices de esas filas
# Luego se eliminan usando drop
# inplace=True modifica directamente el DataFrame original
users_dataset.drop(
    users_dataset[pd.isnull(users_dataset['Age'])].index,
    inplace=True
)

# Mostrar las primeras 4 filas del DataFrame final ya limpio
display(users_dataset.head(4))
```

UserID	Gender	Age	Occupation	Zip-code
1	F	1	10	48067
19	M	1	10	48073
51	F	1	10	10562
75	F	1	10	01748

```
------------------------------------------------

       Gender   Age  Occupation Zip-code
UserID
1           F   NaN          10    48067
2           M  56.0          16    70072
3           M  25.0          15    55117
4           M  45.0           7    02460
5           M  25.0          20    55455
…         …   …             …       …
6036        F  25.0          15    32603
6037        F  45.0           1    76006
6038        F  56.0           1    14706
6039        F  45.0           0    01060
6040        M  25.0           6    11106

[6040 rows x 4 columns]

------------------------------------------------

       Gender  Age  Occupation Zip-code
UserID
1           F  NaN          10    48067
19          M  NaN          10    48073
51          F  NaN          10    10562
75          F  NaN          10    01748

------------------------------------------------

       Gender   Age  Occupation Zip-code
UserID
2           M  56.0          16    70072
3           M  25.0          15    55117
4           M  45.0           7    02460
5           M  25.0          20    55455
```

[183]:
```
# Agrupar los datos del DataFrame por el atributo 'Gender'
# Para cada grupo (por ejemplo 'M' y 'F'),
# se calculan estadísticas descriptivas de las columnas numéricas
display(users_dataset.groupby(by='Gender').describe())
```

```
          Age                                                          Occupation  \
        count       mean        std   min   25%   50%   75%   max      count
Gender
F      1631.0  32.287554  11.792015  18.0  25.0  25.0  45.0  56.0     1631.0
M      4187.0  31.568665  11.716053  18.0  25.0  25.0  35.0  56.0     4187.0


            mean       std  min  25%  50%   75%   max
Gender
```

```
F       6.498467  5.960285  0.0  1.0  4.0  11.0  20.0
M       8.743253  6.441753  0.0  4.0  7.0  15.0  20.0
```

[184]:
```python
# Construir la ruta donde se guardará el archivo CSV de salida
# os.path.join crea una ruta compatible con cualquier sistema operativo
ruta_output = os.path.join('res', 'ml-1m', 'o_users_processed.csv')

# Guardar el DataFrame 'users_dataset' en un archivo CSV
# sep=',' indica que el separador será la coma
# na_rep='null' indica que los valores NaN se escribirán como la cadena 'null'
users_dataset.to_csv(
    ruta_output,
    sep=',',
    na_rep='null'
)
```

2.1 Ejercicios

- Hacer un análisis general de los otros dos archivos CSV en ml-1m ('movies.dat' y 'ratings.dat')
- Analizando el dataset ratings.dat, ¿hay algún usuario que no tenga ninguna review? ¿Cuántos tienen menos de 30 reviews?

[186]:
```python
import os
import pandas as pd

ruta_ratings = os.path.join("res", "ml-1m", "ratings.dat")
# Leer el archivo ratings.dat sin cabecera
# sep='::' indica que el separador es '::'
# header=None indica que el archivo no tiene fila de encabezados
# names asigna los nombres de las columnas manualmente
# engine='python' es necesario porque el separador tiene más de un carácter
ratings_dataset = pd.read_csv(
    ruta_ratings,
    sep='::',
    header=None,
    names=['UserID', 'MovieID', 'Rating', 'Timestamp'],
    engine='python'
)

# Mostrar el DataFrame completo
display(ratings_dataset)
```

```
         UserID  MovieID  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
…           …       …      …         …
```

```
1000204     6040      1091        1  956716541
1000205     6040      1094        5  956704887
1000206     6040       562        5  956704746
1000207     6040      1096        4  956715648
1000208     6040      1097        4  956715569

[1000209 rows x 4 columns]
```

[187]:
```python
# Calcular cuántos usuarios tienen al menos una review (valoración)
# groupby('UserID') agrupa todas las reviews por usuario
# len(...) cuenta cuántos grupos distintos existen
n_users_with_reviews = len(ratings_dataset.groupby(by='UserID'))
print(n_users_with_reviews)

# Calcular el número total de usuarios en el dataset de usuarios
# Cada fila representa un usuario distinto
n_users = len(users_dataset)
print(n_users)

# Calcular cuántos usuarios NO tienen ninguna review
# Se obtiene restando los usuarios con reviews del total de usuarios
print(n_users - n_users_with_reviews)
```

```
6040
5818
-222
```

[188]:
```python
# Agrupar el dataset de ratings por usuario (UserID)
# Luego se filtran solo los usuarios que tienen MENOS de 30 reviews
ratings_under_30 = (
    ratings_dataset
        .groupby(['UserID'])['UserID']
        .filter(lambda x: len(x) < 30)
        .value_counts()
)

# Mostrar cuántas reviews tiene cada usuario que cumple la condición
display(ratings_under_30)

# Mostrar cuántos usuarios distintos tienen menos de 30 reviews
print(len(ratings_under_30))
```

```
UserID
71      29
5814    29
5898    29
2775    29
5909    29
        ..
```

```
2673    20
160     20
5533    20
5525    20
217     20
Name: count, Length: 751, dtype: int64
```

```
751
```

[193]:
```
import os
import pandas as pd

ruta_movies = os.path.join("res", "ml-1m", "movies.dat")

#Le puse el encoding latin-1 para que se pueda visualizar con las tildes, dice␣
 ↪que no soportaba el utf-8
movies_dataset = pd.read_csv(ruta_movies, sep='::', header= None,
                             names = ['MovieID', 'Title', 'Genres'],
                             engine= 'python', encoding='latin-1')

display(movies_dataset)
```

```
      MovieID                                Title  \
0           1                    Toy Story (1995)
1           2                      Jumanji (1995)
2           3             Grumpier Old Men (1995)
3           4            Waiting to Exhale (1995)
4           5  Father of the Bride Part II (1995)
…       …                                    …
3878     3948             Meet the Parents (2000)
3879     3949          Requiem for a Dream (2000)
3880     3950                    Tigerland (2000)
3881     3951             Two Family House (2000)
3882     3952               Contender, The (2000)

                          Genres
0      Animation|Children's|Comedy
1     Adventure|Children's|Fantasy
2                   Comedy|Romance
3                     Comedy|Drama
4                           Comedy
…                            …
3878                        Comedy
3879                         Drama
3880                         Drama
3881                         Drama
3882                Drama|Thriller
```

```
[3883 rows x 3 columns]
```

[194]:
```
#1) GÉNEROS MÁS FRECUENTES

# Separar los géneros y contarlos
#El explode Rompe listas dentro de una celda
#y las expande en varias filas.
genres_count = (
    movies_dataset['Genres']
    .str.split('|')
    .explode()
    .value_counts()
)

display(genres_count)
```

```
Genres
Drama          1603
Comedy         1200
Action          503
Thriller        492
Romance         471
Horror          343
Adventure       283
Sci-Fi          276
Children's      251
Crime           211
War             143
Documentary     127
Musical         114
Mystery         106
Animation       105
Fantasy          68
Western          68
Film-Noir        44
Name: count, dtype: int64
```

[195]:
```
# Contar películas cuyo título contiene el año 1995
movies_1995 = movies_dataset[movies_dataset['Title'].str.contains('(1995)',␣
 ↪regex=False)]

print(len(movies_1995))
```

```
342
```

[198]:
```
# Contar películas con más de un género

#regex=False le dice a Pandas que el texto que estás buscando
```

```
#debe interpretarse literalmente, no como una expresión regular
#En cristiano, que lo tome literalmente por lo que es
#Si no pongo esto, entonces lo va a interpretar como un OR
movies_multi_genre = movies_dataset[
    movies_dataset['Genres'].str.contains('|', regex=False)
]

print(len(movies_multi_genre))
```

```
1858
```

[200]:
```
# Buscar películas sin género (NaN o cadena vacía)
movies_no_genre = movies_dataset[
    movies_dataset['Genres'].isnull() | (movies_dataset['Genres'] == '')
]

display(movies_no_genre)
print(len(movies_no_genre))
#no hay ninguna sin género xd
```

```
Empty DataFrame
Columns: [MovieID, Title, Genres]
Index: []
```

```
0
```

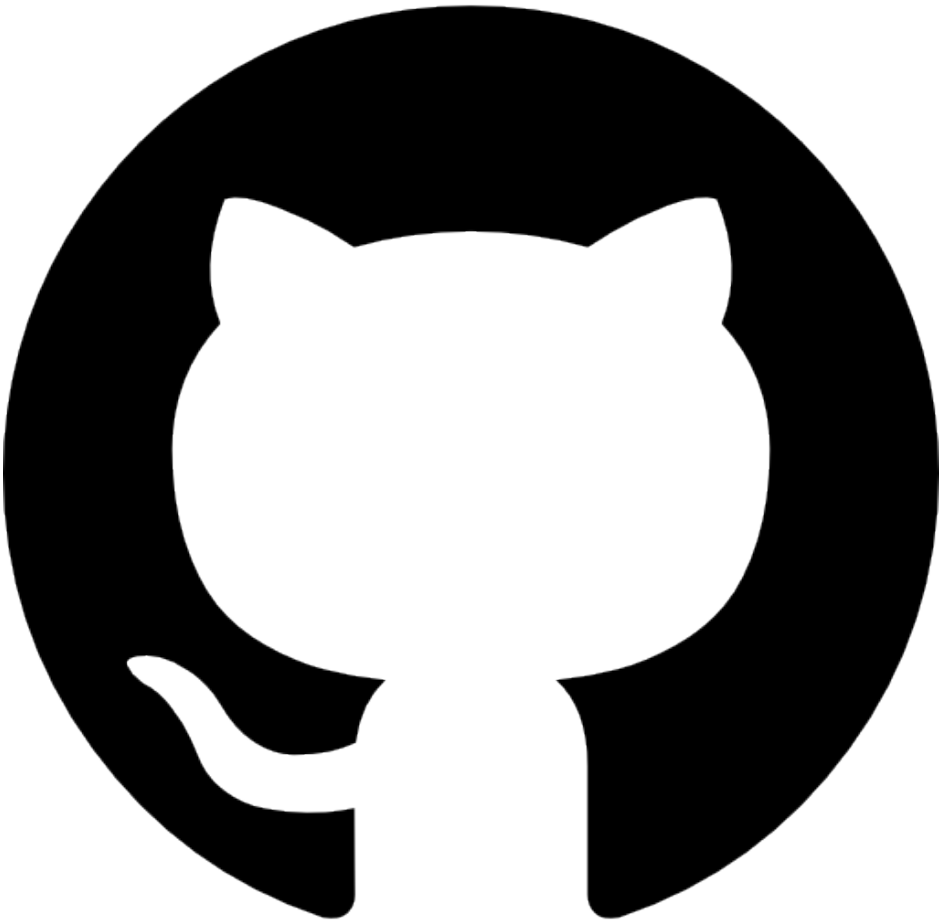
[201]:
```
# Buscar títulos duplicados
duplicate_titles = movies_dataset[movies_dataset['Title'].
 ↪duplicated(keep=False)]

display(duplicate_titles)
print(len(duplicate_titles))
```

```
Empty DataFrame
Columns: [MovieID, Title, Genres]
Index: []
```

```
0
```

[ ]:
```
#Tampoco hay repetidos xd
```

3 Github



3.0.1 Click aquí para ver el repositorio

[ ]: