Python_Pandas

January 20, 2026

0.1 PAO 25-25- Pandas

Gabriel Salguero

0.2 Pandas

- Librería (de facto estándar) para estructurar datos tabulares
- Multivariable (string, int, float, bool…)
- Dos clases:
  - Series (1 dimensión)
  - DataFrames (2+ dimensiones)

[88]:
```
# librería externa
import pandas as pd
from pandas import Series, DataFrame
```

[89]:
```
pd.__version__
```

[89]:
```
'2.3.3'
```

0.3 Series

- Datos unidimensionales (similar a NumPy)
- Elementos + índices modificables

[90]:
```
countries = pd.Series(['Spain','Andorra','Gibraltar','Portugal','France'])
print(countries)
```

```
0        Spain
1      Andorra
2    Gibraltar
3     Portugal
4       France
dtype: object
```

[91]:
```
# especificando el índice
countries = pd.Series(['Spain','Andorra','Gibraltar','Portugal','France'],
                      index=range(10,60,10))
print(countries)
```

```
10        Spain
20      Andorra
30    Gibraltar
40     Portugal
50       France
dtype: object
```

[92]:
```
# los índices pueden ser de más tipos
football_cities = pd.Series(['Barcelona','Madrid','Valencia','Sevilla'],
                            index=['a','b','c','d'])
print(football_cities)
```

```
a    Barcelona
b       Madrid
```

```
c     Valencia
d      Sevilla
dtype: object
```

[93]:
```
# Atributos
football_cities.name = 'Ciudades con dos equipos en primera' # nombrar la Serie
football_cities.index.name = 'Id' # Describir los índices
print(football_cities)
```

```
Id
a    Barcelona
b       Madrid
c     Valencia
d      Sevilla
Name: Ciudades con dos equipos en primera, dtype: object
```

[94]:
```
# acceso similar a NumPy o listas, según posición (depecrated)
print(football_cities.iloc[2])

# acceso a través del índice semántico
print(football_cities['c'])

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

```
Valencia
Valencia
True
```

0.4 Tratamiento similar a ndarray

[95]:
```
# múltiple recolección de elementos
print(football_cities[ ['a','c'] ])
print(football_cities.iloc[ [0, 3] ])
```

```
Id
a    Barcelona
c     Valencia
Name: Ciudades con dos equipos en primera, dtype: object
Id
a    Barcelona
d      Sevilla
Name: Ciudades con dos equipos en primera, dtype: object
```

[96]:
```
# slicing
print(football_cities[:'c']) # incluye ambos extremos con el indice semantico
print(football_cities[:2])
```

```
Id
a    Barcelona
b       Madrid
```

```
c     Valencia
Name: Ciudades con dos equipos en primera, dtype: object
Id
a    Barcelona
b       Madrid
Name: Ciudades con dos equipos en primera, dtype: object
```

[97]:
```
#cast a list
lista = list(football_cities[:'c'])
print(lista)
print(type(lista))
```

```
['Barcelona', 'Madrid', 'Valencia']
<class 'list'>
```

[98]:
```
type(football_cities[:'c'])
```

[98]: pandas.core.series.Series

[99]:
```
#cast a ndarray
import numpy as np
cities = np.array(football_cities[:'c'])
print(cities)
print(type(cities))
```

```
['Barcelona' 'Madrid' 'Valencia']
<class 'numpy.ndarray'>
```

[100]:
```
# cas a dictionary
lista = dict(football_cities[:'c'])
print(lista)
```

```
{'a': 'Barcelona', 'b': 'Madrid', 'c': 'Valencia'}
```

[101]:
```
# uso de masks para seleccionar
fibonacci = pd.Series([0, 1, 1, 2, 3, 5, 8, 13, 21])
print(fibonacci)

mask = fibonacci > 10
print(mask)
print(fibonacci[mask])

dst = pd.Series([13,21])
print(dst)

dst.equals(fibonacci)

fb = fibonacci[mask]
fb.reset_index(drop=True, inplace=True)
```

```
print(fb)

dst.equals(fb)
```

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

[101]:
```
True
```

[102]:
```
# aplicar funciones de numpy a la serie
import numpy as np

print(np.sum(fibonacci))
```

```
54
```

[103]:
```
#filtrado con np.where
distances = pd.Series([12.1,np.nan,12.8,76.9,6.1,7.2])

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'>
```

0.4.1 Iteración

[104]:
```
# iterar sobre elementos
for value in fibonacci:
    print('Value: ' + str(value))

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

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

[105]:
```
# iterar sobre elementos e índices al mismo tiempo
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
```

[106]:
```
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
```

0.4.2 Series como diccionarios

- Interpretar el índice como clave
- Acepta operaciones para diccionarios

[107]:
```
# crear una serie a partir de un diccionario
serie = pd.Series( { 'Carlos' : 100, 'Marcos': 98} )

print(serie.index)
print(serie.values)

print(serie)
print(type(serie))
```

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

[108]:
```
# 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
```

[109]:
```
# query una serie
# print(serie['Marcos'])

if 'Marcos' in serie:
    print(serie['Marcos'])
```

```
print(serie)
```

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

0.4.3 Operaciones entre series

[110]:
```
# suma de dos series
# suma de valores con el mismo índice (NaN si no aparece en ambas)
serie1 = pd.Series([10,20,30,40], index=range(4) )
serie2 = pd.Series([1,2,3], index=range(3) )
suma = serie1 + serie2
print(suma)
```

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

[111]:
```
# resta de series (similar a la suma)
print(serie1- serie2)
```

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

[112]:
```
# operaciones de pre-filtrado
result = serie1 + serie2
result[pd.isnull(result)] = 0 # mask con isnull()
print(result)
```

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

0.4.4 Diferencias entre Pandas Series y diccionario

- Diccionario, es una estructura que relaciona las claves y los valores de forma arbitraria.
- Series, estructura de forma estricta listas de valores con listas de índice asignado en la posición.
- Series, es más eficiente para ciertas operaciones que los dicionarios.
- En las Series los valores de entrada pueden ser listas o Numpy arrays.
- En Series los índices semánticos pueden ser integers o caracteres, en los valores igual.
- Series se podría entender entre una lista y un diccionario Python, pero es de una dimensión.

0.5 DataFrame

- Datos tabulares (filas x columnas)
- Columnas: Series con índices compartidos

[113]:
```
# crear un DataFrame a partir de un diccionario de elementos de la misma␣
 ↪longitud
diccionario = { "Nombre" : ["Marisa","Laura","Manuel"],
                "Edad" : [34,29,12] }
print(diccionario)

# las claves identifican columnas
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

[114]:
```
# crear un DataFrame a partir de un diccionario de elementos de la misma␣
 ↪longitud
diccionario = { "Nombre" : ["Marisa","Laura","Manuel"],
                "Edad" : [34,29,12] }

# las claves identifican columnas
frame = pd.DataFrame(diccionario, index = ['a', 'b', 'c'])
display(frame)
```

	Nombre	Edad
a	Marisa	34
b	Laura	29
c	Manuel	12

[115]:
```
# además de '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	Laura	29	NaN	NaN
2	NaN	Manuel	12	NaN	NaN

[116]:
```
# acceso a columnas
nombres = frame['Nombre']
display(nombres)
```

```
print(type(nombres))

edades = frame['Edad']
display(edades)
print(type(edades))
```

```
0    Marisa
1     Laura
2    Manuel
Name: Nombre, dtype: object

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

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

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

[117]:
```
#siempre que el nombre de la columna lo permita (espacios, ...)
nombres = frame.Profesion
display(nombres)
type(nombres)
```

```
0    NaN
1    NaN
2    NaN
Name: Profesion, dtype: object
```

[117]:
```
pandas.core.series.Series
```

[118]:
```
# acceso al primer nombre del DataFrame frame??
print(frame['Nombre'][0])
print(frame.Nombre[0])
print(nombres[0])
```

```
Marisa
Marisa
nan
```

0.5.1 Formas de crear un DataFrame

- Con una Serie de pandas
- Lista de diccionarios
- Dicionario de Series de Pandas
- Con un array de Numpy de dos dimensiones
- Con array estructurado de Numpy

0.5.2 Modificar DataFrames

[119]:
```
# añadir columnas
diccionario = { "Nombre" : ["Marisa","Laura","Manuel"],
                "Edad" : [34,29,12] }

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

	Nacionalidad	Nombre	Edad	Profesion	Direccion
0	NaN	Marisa	34	NaN	Desconocida
1	NaN	Laura	29	NaN	Desconocida
2	NaN	Manuel	12	NaN	Desconocida

[120]:
```
lista_direcciones = ['Rue 13 del Percebe, 13', 'Evergreen Terrace, 3', 'Av de␣
 ↪los Rombos, 12']
```

[121]:
```
frame['Direccion'] = lista_direcciones

display(frame)
```

	Nacionalidad	Nombre	Edad	Profesion	Direccion
0	NaN	Marisa	34	NaN	Rue 13 del Percebe, 13
1	NaN	Laura	29	NaN	Evergreen Terrace, 3
2	NaN	Manuel	12	NaN	Av de los Rombos, 12

[122]:
```
# añadir fila (requiere todos los valores)
user_2 = ['Alemania','Klaus',20, 'none','Desconocida']
frame.loc[3] = user_2
display(frame)
```

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

[123]:
```
# eliminar fila (similar a Series)
frame = pd.DataFrame(diccionario,columns=['Nacionalidad', 'Nombre',␣
 ↪'Edad','Profesion'])

frame = frame.drop(2) # por qué necesitamos reasignar el frame?
display(frame)

frame.drop('Nombre', axis = 1, inplace = True)
display(frame)
```

	Nacionalidad	Nombre	Edad	Profesion

```
0          NaN  Marisa    34       NaN
1          NaN   Laura    29       NaN
```

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

[124]:
```
#eliminar columna
del frame['Profesion']
display(frame)
```

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

[125]:
```
# acceder a la traspuesta (como una matriz)
display(frame.T)
```

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

0.5.3 Iteración

[126]:
```
# iteración sobre el DataFrame?
frame = pd.DataFrame(diccionario, columns=['Nacionalidad', 'Nombre',␣
 ↪'Edad','Profesion'])
display(frame)

for a in frame:
    print(a) # qué es 'a'?
    print(type(a))
```

```
  Nacionalidad  Nombre  Edad Profesion
0          NaN  Marisa    34       NaN
1          NaN   Laura    29       NaN
2          NaN  Manuel    12       NaN
```

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

[127]:
```
# iteracion sobre filas
for value in frame.values:
    print(value)
```

```
    print(type(value))
```

```
[nan 'Marisa' 34 nan]
<class 'numpy.ndarray'>
[nan 'Laura' 29 nan]
<class 'numpy.ndarray'>
[nan 'Manuel' 12 nan]
<class 'numpy.ndarray'>
```

[128]:
```
# iterar sobre filas y luego sobre cada valor?
for values in frame.values:
    for value in values:
        print(value)
```

```
nan
Marisa
34
nan
nan
Laura
29
nan
nan
Manuel
12
nan
```

0.5.4 Indexación y slicing con DataFrames

[129]:
```
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] # lista de diccionarios
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

[130]:
```
# Acceso a un valor concreto por indice posicional [row, col]
print(df_data.iloc[1,1])

# Acceso a todos los valores hasta un índice por enteros
display(df_data.iloc[:3,:4])

# Acceso a datos de manera explícita, indice semantico (se incluyen)
display(df_data.loc['a', 'temperatura'])
display(df_data.loc[:'c', :'o2'])
display(df_data.loc[:'c', 'temperatura':'o2'])

display(df_data.loc[:, ['ciudad','o2']])
```

```
8
```

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

```
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

[131]:
```
# indexación con nombre de columna (por columnas)
print(df_data['ciudad']) #--> Series

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
```

[132]:
```
# indexación con índice posicional (no permitido!). Esto busca columna.
# df_data[0] # Descomentar para ver el error KeyError
```

[133]:
```
# indexar por posición con 'iloc'
print(df_data.iloc[0]) #--> Series de la primera fila (qué marca los índic
```

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

[134]:
```
# indexar semántico con 'loc'
df_data.loc['a'] #--> Series de la fila con índice 'a'
```

[134]:
```
ciudad         Valencia
temperatura          10
o2                  1.0
humedad             NaN
co2                 NaN
Name: a, dtype: object
```

[135]:
```
# indexar semántico con 'loc'
df_data.loc[:'b'] #--> DataFrame de la fila con índice 'a' y 'b'
```

[135]:

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

[136]:
```
df_data.loc[:'b'].loc[:,["o2", "humedad"]] # slicing anidado
```

[136]:

	o2	humedad
a	1.0	NaN
b	NaN	NaN

[137]:
```
# si se modifica una porcion del dataframe se modifica el dataframe original
# (referencia)
```

```
display(df_data)

serie = df_data.loc['a']
print(serie)
serie.iloc[2] = 3000 # setting with copy warning!!!
display(df_data)

# copiar data frame
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\Reromash\AppData\Local\Temp\ipykernel_14936\3020112947.py:7:
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 # setting with copy warning!!!
```

```
      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                 3000
humedad             NaN
co2                 NaN
```

```
Name: a, dtype: object
```

	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

[138]:
```
# ambos aceptan 'axis' como argumento
# df_data.iloc(axis=1)[0] #--> todos los valores asignados a la primera columna␣
  ↪'ciudad'
# df_data.loc(axis=1)['ciudad'] #--> equivalente frame['ciudad']
```

[139]:
```
# 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

[140]:
```
# 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
```

0.5.5 Objeto Index de Pandas

[141]:
```
# Contrucción 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')
```

[142]:
```
# 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
```

[143]:
```
# Son inmutables! No se modifican los datos.
# ind[3] = 8
```

[144]:
```
# change index column
frame = pd.DataFrame({"Name" : ['Carlos','Pedro', 'Manolo', 'Luis', 'Alberto'],␣
 ↪"Age" : [34,22,15,55,23]}, index=ind)
display(frame)

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
```

0.5.6 Slicing

[145]:
```
# slice por filas
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'])
```

```
display(character_data)
display(character_data[:-1])
display(character_data[1:2])
```

```
       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
```

[146]:
```
# slicing para columnas
display(character_data[['Name','Wisdom']])
```

```
       Name  Wisdom
a  bundenth      20
b   theorin      13
c    barlok       6
```

[147]:
```
#slicing con 'loc' e 'iloc'
display(character_data.iloc[1:])
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'

[148]:
```
# usando 'loc' para hacer slicing
display(character_data.loc[:,'Name':'Strength'])
```

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

[149]:
```
# usando 'loc' para buscar específicamente filas y columnas
display(character_data.loc[ ['a','c'], ['Name','Wisdom'] ])
```

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

[150]:
```
# lo mismo con 'iloc'?
display(character_data.iloc[[0,2],[0,2]])
display(character_data.iloc[[0,-1],[0,-1]])
```

	Name	Wisdom
a	bundenth	20
c	barlok	6

	Name	Wisdom
a	bundenth	20
c	barlok	6

[151]:
```
# 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

[152]:
```
# 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

0.6 Cargar y guardar datos en pandas

[153]:
```
# Guardar a csv
import os
ruta = os.path.join("res" ,"o_d_d_characters.csv")

#character_data.to_csv(ruta, sep=';') # sep por defecto: ','
```

[154]:
```
loaded = pd.read_csv(ruta, sep=';')
display(loaded)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[154], line 1
----> 1 loaded = pd.read_csv(ruta, sep=   )
      2 display(loaded)
```

```
File E:\anaconda3\envs\machines2026\Lib\site-packages\pandas\io\parsers\readers
 ↪py:1026, in read_csv(filepath_or_buffer, sep, delimiter, header, names,␣
 ↪index_col, usecols, dtype, engine, converters, true_values, false_values,␣
 ↪skipinitialspace, skiprows, skipfooter, nrows, na_values, keep_default_na,␣
 ↪na_filter, verbose, skip_blank_lines, parse_dates, infer_datetime_format,␣
 ↪keep_date_col, date_parser, date_format, dayfirst, cache_dates, iterator,␣
 ↪chunksize, compression, thousands, decimal, lineterminator, quotechar,␣
 ↪quoting, doublequote, escapechar, comment, encoding, encoding_errors, dialect␣
 ↪on_bad_lines, delim_whitespace, low_memory, memory_map, float_precision,␣
 ↪storage_options, dtype_backend)
   1013 kwds_defaults = _refine_defaults_read(
   1014     dialect,
   1015     delimiter,
   (…)   1022     dtype_backend=dtype_backend,
   1023 )
   1024 kwds.update(kwds_defaults)
-> 1026 return _read(filepath_or_buffer, kwds)

File E:\anaconda3\envs\machines2026\Lib\site-packages\pandas\io\parsers\readers
 ↪py:620, in _read(filepath_or_buffer, kwds)
    617 _validate_names(kwds.get("names", None))
    619 # Create the parser.
--> 620 parser = TextFileReader(filepath_or_buffer, **kwds)
    622 if chunksize or iterator:
    623     return parser

File E:\anaconda3\envs\machines2026\Lib\site-packages\pandas\io\parsers\readers
 ↪py:1620, in TextFileReader.__init__(self, f, engine, **kwds)
   1617     self.options["has_index_names"] = kwds["has_index_names"]
   1619 self.handles: IOHandles | None = None
-> 1620 self._engine = self._make_engine(f, self.engine)

File E:\anaconda3\envs\machines2026\Lib\site-packages\pandas\io\parsers\readers
 ↪py:1880, in TextFileReader._make_engine(self, f, engine)
   1878     if "b" not in mode:
   1879         mode += "b"
-> 1880 self.handles = get_handle(
   1881     f,
   1882     mode,
   1883     encoding=self.options.get(          , None),
   1884     compression=self.options.get(             , None),
   1885     memory_map=self.options.get(            , False),
   1886     is_text=is_text,
   1887     errors=self.options.get(                 ,         ),
   1888     storage_options=self.options.get(                , None),
   1889 )
   1890 assert self.handles is not None
   1891 f = self.handles.handle
```

```
File E:\anaconda3\envs\machines2026\Lib\site-packages\pandas\io\common.py:873,␣
 ↪in get_handle(path_or_buf, mode, encoding, compression, memory_map, is_text,␣
 ↪errors, storage_options)
    868 elif isinstance(handle, str):
    869     # Check whether the filename is to be opened in binary mode.
    870     # Binary mode does not support 'encoding' and 'newline'.
    871     if ioargs.encoding and "b" not in ioargs.mode:
    872         # Encoding
--> 873         handle = open(
    874             handle,
    875             ioargs.mode,
    876             encoding=ioargs.encoding,
    877             errors=errors,
    878             newline=  ,
    879         )
    880     else:
    881         # Binary mode
    882         handle = open(handle, ioargs.mode)

FileNotFoundError: [Errno 2] No such file or directory: 'res\\o_d_d_characters.
 ↪csv'
```

[ ]:
```
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)
```

otros argumentos to_csv()

- na_rep='string'–> representar valores NaN en el archivo csv

otros argumentos read_csv()

- na_values='string'

Pandas también ofrece funciones para leer/guardar a otros formatos estándares: JSON, HDF5 o Excel en su API

0.7 Ejemplo práctico en pandas

- MovieLens dataset
- Reviews de películas
- 1 millón de entradas
- Datos demográficos de usuarios

[155]:
```
import numpy as np
import pandas as pd
import zipfile # para descomprimir archivos zip
import urllib.request # para descargar de URL
import os

# 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)
```

[155]:
```
('res\\ml-1m.zip', <http.client.HTTPMessage at 0x1d4c5dc40d0>)
```

[156]:
```
# descomprimiendo archivo zip
ruta_ext = os.path.join("res")
with zipfile.ZipFile(ruta, 'r') as z:
    print('Extracting all files...')
    z.extractall(ruta_ext) # destino
    print('Done!')

# take a look at readme y revisar formatos
```

```
Extracting all files…
Done!
```

[157]:
```
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]
```

[158]:
```
# Varios problemas
# sin cabecera! primer valor se ha perdido
# las columnas no tienen nombres
```

```
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:      e:
 ↪\anaconda3\envs\machines2026\lib\site-packages\pandas\io\parsers\readers.py
Type:      function
```

[159]:
```
# 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]
```

[160]:
```
# samplear la tabla
display(users_dataset.sample(10))
```

```
       Gender  Age  Occupation Zip-code
UserID
1557        F   18           4    10039
3925        M   25          12    77042
757         M   25          12    97401
4073        M   25           6    70003
922         M   56          16    48009
4270        M   45           7    13211
3171        F   50           3    10128
3249        F   25           4    92648
1138        M   18          12    12047
684         M   25           4    27510
```

[161]:
```
# samplear la cabeza
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
```

[162]:
```
# samplear la cola
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
```

[163]:
```
# tipos de datos sobre las columnas
users_dataset.dtypes
```

[163]:
```
Gender        object
Age            int64
Occupation     int64
Zip-code      object
dtype: object
```

[164]:
```
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]
```

[165]:
```
# 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
```

[166]:
```
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
```

[167]:
```
# 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
```

[168]:
```
# cuántos usuarios son mujeres (Gender='F')
len(users_dataset[users_dataset['Gender'] == 'F'])

#select count(*) from users_dataset where users_dataset.Gender = 'F'
```

[168]: 1709

[169]:
```
# mostrar solo los menores de edad
under_age = users_dataset[users_dataset['Age'] == 1]
print(len(under_age))
display(under_age.sample(10))
```

```
222
```

```
        Gender  Age  Occupation Zip-code
UserID
906          M    1          10    71106
1195         F    1          10    13907
1365         F    1          10    61665
4294         M    1          10    75633
1509         M    1           0    11803
5302         F    1          10    02332
3190         M    1          17    97062
1241         M    1          10    11803
5803         M    1          10    01597
6006         F    1           0    01036
```

[170]:
```
# filtrar edad incorrecta (mininimo 18) SettingWithCopyWarning!!!
users_dataset = pd.read_csv(ruta_users, sep='::', index_col=0,
    header=None, names=['UserID','Gender','Age','Occupation','Zip-code'],␣
 ↪engine='python')
under_age = users_dataset[users_dataset['Age'] == 1]

under_age.loc['Age'] = np.nan
display(under_age.head())

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

```
C:\Users\Reromash\AppData\Local\Temp\ipykernel_14936\2304130102.py:6:
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
  under_age.loc['Age'] = np.nan
```

UserID	Gender	Age	Occupation	Zip-code
1	F	1.0	10.0	48067
19	M	1.0	10.0	48073
51	F	1.0	10.0	10562
75	F	1.0	10.0	01748
86	F	1.0	10.0	54467

[171]:
```
# filtrar edad incorrecta (mínimo 18) Fixing it by Copying the slice
users_dataset = pd.read_csv(ruta_users, sep='::', index_col=0,
    header=None, names=['UserID','Gender','Age','Occupation','Zip-code'],␣
 ↪engine='python')
under_age = users_dataset[users_dataset['Age'] == 1]

under_age_copy = under_age.copy()
display(under_age_copy.head())

under_age_copy['Age'] = np.nan
display(under_age_copy.head())

users_dataset[users_dataset['Age'] == 1] = under_age_copy # sets the rows␣
 ↪according to the indexes
display(users_dataset.head())
```

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
86	F	1	10	54467

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

	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
```

[172]:
```
# filtrar edad incorrecta (mínimo 18) Remove them from the dataset
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[users_dataset['Age'] == 1].head(4))

users_dataset.loc[users_dataset['Age'] == 1, 'Age'] = np.nan
display(users_dataset)

display(users_dataset.loc[pd.isnull(users_dataset['Age'])].head(4))

users_dataset.drop(users_dataset[pd.isnull(users_dataset['Age'])].index,␣
 ↪inplace = True)
display(users_dataset.head(4))
```

```
       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
```

```
       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
```

[173]:
```
# Agrupar datos por atributos
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
```

[174]:
```
# Grabar la tabla modificada
# Cambiar el separador a ','
# Guardar NaN como 'null'
ruta_output = os.path.join('res', 'ml-1m', 'o_users_processed.csv')
users_dataset.to_csv(ruta_output, sep=',',na_rep='null')
```

0.8 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?

[175]:
```
# Cargar ratings.dat
# especificar nombres, cargar sin cabecera
import os
import pandas as pd
ruta_ratings = os.path.join("res", "ml-1m", "ratings.dat")

# Usamos nombres descriptivos para las columnas según readme
col_names = ['UserID', 'MovieID', 'Rating', 'Timestamp']
df_rates = pd.read_csv(ruta_ratings, sep='::', header=None, names=col_names,␣
 ↪engine='python')
```

```
print("Vista previa de ratings:")
display(df_rates.head())
```

```
Vista previa de ratings:
```

	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

[176]:
```
# ¿Hay algún usuario que no tenga ninguna review?
# Usamos conjuntos para comparar IDs presentes en usuarios vs ratings
usuarios_registrados = set(users_dataset.index)
usuarios_activos = set(df_rates['UserID'].unique())

sin_actividad = usuarios_registrados - usuarios_activos

print(f"Total usuarios: {len(usuarios_registrados)}")
print(f"Usuarios con ratings: {len(usuarios_activos)}")
print(f"Usuarios sin reviews: {len(sin_actividad)}")

if len(sin_actividad) > 0:
    print("Sí, hay usuarios inactivos.")
else:
    print("Todos los usuarios tienen reviews.")
```

```
Total usuarios: 5818
Usuarios con ratings: 6040
Usuarios sin reviews: 0
Todos los usuarios tienen reviews.
```

[177]:
```
# ¿Cuántos tienen menos de 30 reviews?
# Calculamos conteo por usuario y filtramos
conteo_reviews = df_rates['UserID'].value_counts()
menos_30 = conteo_reviews[conteo_reviews < 30]

print(f"Usuarios con menos de 30 reviews: {len(menos_30)}")
display(menos_30.head())
```

```
Usuarios con menos de 30 reviews: 751
```

```
UserID
2107    29
2758    29
2775    29
3276    29
5814    29
Name: count, dtype: int64
```

0.9 repostorio

GitHub: https://github.com/reromash1/machines2026

[ ]: