Python: Pandas

Series: estructuras en una dimension

Crear series

`serie = pd.Series()` crear serie vacía
`serie = pd.Series(array)` crear serie a partir de un array con el indice por defecto
`serie = pd.Series(array, index = ['a', 'b', 'c'...])` crear una serie con indice definida; debe ser lista de la misma longitude del array
`serie = pd.Series(lista)` crear una seria a partir de una lista
`serie = pd.Series(número, indice)` crear una serie a partir de un escalar con la longitude igual al número de indices
`serie = pd.Series(diccionario)` crear una serie a partir de un diccionario

Acceder a informacion de una serie

`serie.index` devuelve los indices
`serie.values` devuelve los valores
`serie.shape` devuelve la forma (no. filas)
`serie.size` devuelve el tamaño
`serie.dtypes` devuelve el tipo de dato

`serie[i]` devuelve el valor del elemento en indice i
`serie[[i,j]]` devuelve el valor de los dos elementos
`serie[i:m]` devuelve el valor de un rango

`serie["etiqueta"]` devuelve el valor de los elementos en indices i y j

Operaciones con series

`serie1 +-*/ serie2` suma/resta/multiplica/divide las filas con indices comunes entre las dos series
`serie1.add(serie2, fill_value = número)` suma las filas con indices comunes, y suma el fill value a los valores sin indice comun
`serie1.sub(serie2, fill_value = número)` restan las filas de la seria2 de la serie1 cuando tienen indices comunes, y resta el fill value de las otras indices de serie1
`serie1.mul(serie2, fill_value = número)` multiplica las filas con indices comunes y multiplica el fill value con las otras *usar 1 para conservar el valor*
`serie1.mul(serie2, fill_value = número)` divida las filas de la serie1 entre las de la serie2 cuando tienen indices comunes, y divide las otras por el fill value
`serie1.mod(serie2, fill_value = número)` devuelve el modulo (division sin resta)
`serie1.pow(serie2, fill_value = número)` calcula el exponencial
`serie1.ge(serie2)` compara si serie1 es mayor que serie2 y devuelve True o False
`serie1.le(serie2)` compara si serie1 es menor que serie2 y devuelve True o False

Filtrado booleanos

`serie < > >= <= == valor` devuelve True o False segun si cada condición cumple la condición
`serie1[serie1 < > >= <= == valor]` devuelve solo los valores que cumplen la condición
`np.nan` crear valor nulo (NaN)
`serie.isnull()` devuelve True o False segun si los valores existen o son nulos ("" no cuenta como nulo)
`serie.notnull()` devuelve True o False segun si los valores existen o son nulos ("" no cuenta como nulo)

DataFrames

Crear DataFrames

`df = pd.DataFrame(data, index, columns)`
data: NumPy Array, diccionario, lista de diccionarios
index: indice que por defecto se asigna como 0-(n-1), n siendo el número de filas;
`index = [lista]` para asignar "etiquetas" (nombres de filas)
column: nombre de las columnas; por defecto 0-(n-1);
`columns =[lista]` para poner mas nombres

`df = pd.DataFrame(array)` crear un dataframe a partir de un array con indices y columnas por defecto
`df = pd.DataFrame(diccionario)` crear un dataframe a partir de un diccionario – los keys son los nombres de las columnas

DataFrames: carga de datos

Carga de datos

`df = pd.read_csv("ruta/nombre_archivo.csv")` crear un dataframe de un archivo de Comma Separated Values
`df = pd.read_csv("ruta/nombre_archivo", sep= ";")` crear un dataframe de un csv si el separador es ;
`df = pd.read_csv("ruta/nombre_archivo", index_col= 0)` crear un dataframe de un csv si el archivo ya tiene una columna indice

`df = pd.read_excel("ruta/nombre_archivo.xlsx")` crear un dataframe de un archivo de Excel
- si sale "ImportError:... openpyxl...", en el terminal:
pip3 install openpyxl o **pip install openpyxl**

`df = pd.read_json("ruta/nombre_archivo.json")` crear un dataframe de un archivo de JavaScript Object Notation (formato crudo)
`df = df['data'].apply(pd.Series)` convertir el dataframe de json en un formato legible

`df = pd.read_clipboard(sep='\t')` crear un dataframe de datos en forma de dataframe en el clipboard; el separador podria ser \n ; , etc.

```
with open('ruta/nombre_archivo.pkl', 'wb') as f:
    pickle.dump(df,f)
```
pone los datos de un dataframe en el archivo pkl

`pd.read_pickle('ruta/nombre_archivo.csv').head(n)` leer n filas y 5 columnas del archivo pickle

`pd.read_parquet('ruta/nombre_archivo.parquet')` leer un archivo parquet

Guardado de datos

`df.to_csv('ruta/nombre_archivo.csv')` guardar dataframe como archivo csv
`df.to_excel('ruta/nombre_archivo.xlsx')` guardar dataframe como archivo de Excel
`df.to_json('ruta/nombre_archivo.json')` guardar dataframe como archivo de JSON
`df.to_parquet('ruta/nombre_archivo.parquet')` guardar dataframe como archivo de parquet
`df.to_pickle('ruta/nombre_archivo.pkl')` guardar dataframe como archivo de pickle

ExcelWriter

```
with pd.ExcelWriter("ruta/archivo.ext") as writer:
    df.to_Excel(writer, nombre_hoja = 'nombre')
```
guardar un dataframe en una hoja de Excel

Metodos de exploracion

`df.head(n)` devuelve las primeras n lineas del dataframe
`df.tail(n)` devuelve las últimas n lineas del dataframe

`df.sample(n)` devuelve n filas aleatorias de nuestro dataframe, o uno por defecto

`df.shape` devuelve el número de filas y columnas

`df.dtypes` devuelve el tipo de datos que hay en cada columna

`df.columns` devuelve los nombres de las columnas

`df.describe` devuelve un dataframe con un resumen de los principales estadísticos de las columnas numéricas

`df.info()` devuelve un resumen sobre el no. de columnas, nombres de columnas, numero de valores no nulos y los tipos de datos `df["nombre_columna"].unique()` o `df.nombre_columna.unique()` devuelve un array con los valores únicos de la columna

`df["nombre_columna"].value_counts()` o `df.nombre_columna.value_counts()` devuelve una serie con el recuento de valores únicos en orden descendente

`df.duplicated().sum()` devuelve el numero de filas duplicadas

Eliminar filas duplicadas

`df.drop_duplicates(inplace = True, ignore_index=True)` elimina filas duplicadas; ignore_index para no tener el indice en cuenta

Metodos de estadistica

`df['columna'].mean() | mode() | median() | var() | std()` calcula la media/moda/mediana/variación/desviación estándar de los valores de una columna
`df['columna1'].corr(df['columna2']` calcula la correlacion entre dos variables
`matriz_correlacion = df.corr()` crea una matriz mostrando las correlaciones entre todos los variables
`df_crosstab = pd.crosstab(df['columna1'], df['columna2'], normalize = True, margins = True)`
normalize muestra los valores en porcentajes (por uno)
margins muestra los totales y subtotales
`media_ponderada = np.average(df['columna'], weights = w)` calcula la media ponderada según los pesos
`percentil_n = np.percentile(df['columna'], n)` saca el valor en el percentil n
`q3, q1 = np.percentile(df["columna"], [75, 25])` saca los tercer y primer cuartiles

Sidetable: frecuencias de datos

`df.stb.freq(['columna'])` devuelve un dataframe con informacion sobre la frecuencia de ocurrencia de cada categoría de un variable categorica
parametros:
thresh = n limita los valores mostrados a los más frecuentes hasta un umbral de n% cumulative y agrupando los restantes bajo la etiqueta "other"
other_label = 'etiqueta' cambia la etiqueta 'other'
`value = 'columna'` ordena los resultados por la columna especificada
`df.stb.freq(['columna1', 'columna2'])` combina dos columnas y devuele las frecuencias de las subcategories

`df.stb.missing(['columna']` devuelve informacion sobre la frequencia de datos nulos

Tipos de datos

Tipos de datos en Pandas:
- object
- int64
- float64
- datetime, timedelta[ns]
- category
- bool

`df.dtypes` devuelve el tipo de datos que hay en cada columna
`df_tipo = df.select_dtypes(include = "tipo")` crea un dataframe de las columnas del tipo de datos especificado
`df['columna'] = df['columna'].astype('tipo', copy = True, errors = 'ignore')` convierte una columna en el tipo de dato especificado
copy = True devuelve una copia
copy = False *cuidado: los cambios en los valores pueden propagarse a otros objetos pandas*
errors = ignore omita excepciones; en caso de error devuelve el objeto original
errors = raise permite que se generen excepciones

`pd.options.display.max_columns = None` ejecutar antes del df.head() para poder ver todas las columnas

`pd.set_option("display.precision", 2)`

Outliers

Calcular tres desviaciones estandares:

```
media = df.column.mean()
desviacion = df.column.std()

lcb = media – desviacion * 3
ucb = media + desviacion * 3
```

Eliminar Outliers

`outlier_step = 1.5 * IQR` calcular outlier step
`outliers_data = df[(df['columna'] < Q1 – outlier_step) | (df['columna'] > Q3 + outlier_step)]` identificar datos fuera del rango del maximo hasta el minimo
`lista_outliers_index = list(outliers_data.index)` crear una lista de los indices de las filas con outliers

```
if outliers_data.shape[0] > 0:
    dicc_indices[key] =
(list(outliers_data.index))
```
crear un diccionario de los indices de las filas con nulos; se puede hacer iterando por columnas

`valores = dicc_indices.values()` sacar todos los valores e.g. todos los indices
`valores = {indice for sublista in valores for indice in sublista}` set comprehension para eliminar duplicados
`df_sin_outliers = df.drop(df.index[list (valores)])` crear nuevo dataframe sin outliers

Reemplazar Outliers

```
for k, v in dicc_indices.items():
    media = df[k].mean()
    for i in v:
        df.loc[i,k] = media reemplazar
outliers por la media
```

Valores nulos

Identificar nulos

`df.isnull()` o `df.isna()` devuelve True o False según si cada valor es nulo o no
`df.isnull().sum()` o `df.isna().sum()` devuelve una serie con el número de valores nulos por columnas
`df_%_nulos = ((df.isnull().sum() / df.shape[0] * 100).reset_index()`
`df_%_nulos.columns = ['columna', '% nulos']` crea un dataframe de los porcentajes de los valores nulos

Eliminar nulos

`df.dropna(inplace = True, axis=b, subset=[lista_de_columnas], how=)` quitar nulos
how = 'any' | 'all' por defecto 'any': si hay algun valor NA, se elimina la fila o columna; all: si todos los valores son NA, se elimina la fila o columna
subset una columna o lista de columnas

Tipos de nulos

np.nan significa "not a number"; es un tipo numérico
None valores nulos en columnas tipo string
NaT valores nulos tipo datetime
valores texto: "n/a", "NaN", "nan", "null" strings que normalmente se convierten automaticamente a np.nan
99999 o **00000** integers que se pueden convertir a nulos

Reemplazar nulos

`df = pd.read_csv('archivo.csv', na_values = ['n/a']) .fillna(np.nan)` reemplaza los strings 'n/a' con np.nan al cargar el dataframe

`df.fillna(df[value=n, axis=b, inplace=True)` reemplazar todos los NaN del dataframe con el valor que especifiquemos
`df['columna'].fillna(df['columna'].median, axis=b, inplace=True)` reemplazar los nulos de una columna por la mediana de esa columna
value=n por defecto NaN; es el valor por el que queremos reemplazar los valores nulos que puede ser un escalar, diccionario, serie o dataframe
axis por defecto **0** (filas)
`df.replace(valor_nulo, valor_nuevo, inplace=True, regex=False)` reemplazar los nulos por el valor nuevo

Imputacion de nulos

from sklearn.impute import SimpleImputer
`imputer = SimpleImputer(strategy='mean', missing_values = np.nan)` inicia la instancia del metodo, especificando que queremos reemplazar los nulos por la media
`imputer = imputer.fit(df['columna1'])` aplicamos el imputer
`df['media_columna1'] = imputer.transform(df[['price']])` rellena los valores nulos segun como hemos especificado
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
`imputer = IterativeImputer(n_nearest_features=n, imputation_order='ascending')` crea la instancia
n_nearest_features por defecto None; numero de columnas a utilizar para estimar los valores nulos
imputation_order por defecto ascendente; el orden de imputacion
`imputer.fit(df_numericas)` aplicamos el imputer
`df_datos_trans = pd.DataFrame(imputer.transform(df_numericas), columns = df_numericas.columns)` crea un dataframe de los datos transformados; metemos estas columnas en el dataframe original
from sklearn.impute import KNNImputer
imputerKNN = KNNImputer(n_neighbors=5) crea la instancia
imputerKNN.fit(df_numericas)
`df_knn_imp = pd.DataFrame(imputerKNN.transform(df_numericas), columns = numericas.columns)` crea un dataframe de los datos transformados; metemos estas columnas en el dataframe original

Pandas

Union de datos

.concat() unir dataframes con columnas en comun
`df_union = pd.concat([df1, df2, df3], axis=b, join = 'inner/outer', ignore_index = True/False)`
parametros:
`axis = 0` une por columnas – los dataframes van uno encima del otro; las columnas tienen que ser de formatos compatible
`axis = 1` une por filas – los dataframes van uno al lado del otro; los datos deben ser relacionados para que tenga sentido
`join = 'inner'` solo se quedan elementos que aparecen en todos los dataframes
`join = 'outer'` se queda todo los datos de todos los dataframes
`ignore_index = True/False` por defecto es False; si es True no usa las índices para la union (por ejemplo para union por el axis 0)

.merge() unir las columnas de un dataframe a otro
`df_nuevo = df1.merge(df2, on = 'columna')` inner merge
`df_nuevo = pd.merge(left = df1, right = df2, how='left', left_on = 'columna_df1', right_on = 'columna_df2')` left merge
parametros:
`how = 'left' | 'right' | 'outer' | 'inner' | 'cross'`
`on = columna | [columna1, columna2, etc]` si las columnas se llaman igual en los dos dataframes
`left_on = columna_df1 | right_on = columna_df2` para especificar por donde hacer el merge
`suffixes = ['left', 'right']` por defecto nada, el sufijo que aparecera en columnas duplicadas

.join() unir dataframes por los indices
`df_nuevo = df1.join(df2, on = 'columna', how = 'left')` inner merge
parametros:
`how = 'left' | 'right' | 'outer' | 'inner'` por defecto left
`on = columna` la columna o indice por el que queremos hacer el union; tienen que tener el mismo nombre en los dos dataframes
`lsuffix = 'string' | rsuffix = 'string'` por defecto nada, el sufijo que aparecera en columnas duplicadas

Group By

`df_groupby = df.groupby("columna_categoría")` crea un objeto DataFrameGroupBy; agrupa los valores segun las categorías de los valores de la columna indicada (o múltiples columnas en una lista)
`df_groupby.ngroups` devuelve el numero de grupos
`df_groupby.groups` devuelve un diccionario donde los keys son las categorías y los valores son listas de los índices de cada elemento en la categoría
`df_grupo1 = df_groupby.get_group("grupo1")` devuelve un dataframe con los resultados de un grupo (la categoria indicada como grupo1)

Cálculos con groupby:

`df_nuevo = df.groupby("columna_categoría").mean()` devuelve un dataframe con la media de todas las columnas de valores numéricos, por categoría
`df_nuevo = df.groupby("columna_categoría")["columna1"].mean()` devuelve un dataframe con la media de la columna especificada
`count()` número de observaciones no nulas
`describe()` resumen de los principales estadísticos
`sum()` suma de todos los valores
`mean()` media de los valores
`df_nuevo = df.groupby("columna_categoría", dropna = False)["columna_valores"].agg([nombre_columna = 'estadistico1', nombre_columna2 = 'estadistico2'])` añade columnas con los cálculos de los estadísticos especificados
`dropna = False` para tener en cuenta los Nan en los cálculos (por defecto es True)

Subsets: loc e iloc

`df.loc["etiqueta_fila", "etiqueta_columna"]` devuelve el contenido de un campo en una columna de una fila

`df.loc["etiqueta_fila",:]` devuelve los valores de todas las columnas de una fila

`df.loc[:,"etiqueta_columna"]` devuelve los valores de todas las filas de una columna

`df.iloc[indice_fila, indice_columna]` devuelve el contenido de un campo en una columna de una fila

`df.iloc[indice_fila, :]` devuelve los valores de todas las columnas de una fila

`df.iloc[:,indice_columna]` devuelve el contenido de un campo en una columna de una fila

`df.loc[[lista_etiquetas_filas], [lista_etiquetas_columnas]]` devuelve el contenido de varias filas / varias columnas

`df.loc[[lista_indices_filas], [lista_indices_columnas]]` devuelve el contenido de varias filas / varias columnas

- se puede usar los indices/rangos de las listas [start:stop:step] dentro de los loc/iloc

`df.loc[df.etiqueta > x]` seleccionar datos basado en una condición usando operadores comparativos

`df.loc[(df.etiqueta > x) & (df.etiqueta == y)]` seleccionar datos que tienen que cumplir las dos condiciónes (and)

`df.loc[(df.etiqueta > x) | (df.etiqueta == y)]` seleccionar datos que tienen que deben cumplir una de las dos condiciones (or)

`df.iloc[list(df.etiqueta > x), :]` iloc no acepta una Serie booleana; hay que convertirla en lista

`variable_df.head(n)` devuelve las n primeras filas del df, o 5 por defecto

Filtrados de datos

Filtrado por una columna con operadores de comparación

`df_filtrado = df[df["nombre_columna"] == valor]` extrae las filas donde el valor de la columna igual al valor dado

Filtrado por multiples columnas con operadores logicos

`df_filtrado = df[(df["columna1"] == valor) & (df["columna2"] == valor) & (df["columna3"] > n valor)]` extrae las filas donde los valores de las columnas cumplan las condiciónes en parentesis

`df_filtrado = df[(df["columna1"] == valor) | (df["columna1"] == valor)` extrae las filas donde los valores de las columnas cumplan con una condición u otra

`df_filtrado = ~(df[df["columna1"] == valor])` extrae las filas donde los valores de las columnas NO cumplan con la condición

Filtrados de datos

Metodos de pandas de filtrar

`df_filtrado = df[df["nombre_columna"].isin(iterable)]` extrae las filas cuyas valores de la columna nombrada están en el iterable (una lista, serie, dataframe o diccionario)

`df_filtrado= df[df["nombre_columna"].str.contains (patron, regex = True, na = False)]` extrae las filas cuyas valores de la columna nombrada contenienen el patron de regex

`df_filtrado = df[df["nombre_columna"].str.contains ("substring", case = False, regex = False)]` extrae las filas cuyas valores de la columna nombrada contienen el substring, no siendo case sensitive

`df_filtrado = df[df["nombre_columna"].str.contains ("substring", case = False, regex = False)]` extrae las filas cuyas valores de la columna nombrada contienen el substring, no siendo case sensitive

`df[pd.notnull(df["nombre_columna"])]` devuelve las filas que no tiene valores nulos en la columna especificada

Cambiar columnas

`lista_columnas = df.columns.to_list()` crea una lista de los nombres de las columnas del dataframe

`df.set_index(["nombre_columna"], inplace = True)` establece el índice utilizando uno o mas columnas; puede sustituir o ampliar un índice existente
`inplace = True` los cambios sobreescriben sobre el df

*** cuando una columna se cambia a índice ya no es columna ***

`df.reset_index(inplace = True)` quitar una columna como indice para que vuelva a ser columna; crea un dataframe de una serie

Renombrar columnas

`df.rename(columns = {"nombre_columna": "nombre_nueva"}, inplace = True)` cambia los nombres de una o mas columnas

ejemplo de dict comprehension para crear diccionario sobre las columnas existentes de un dataframe:
`diccionario = {col : col.upper() for col in df.columns}`
`df.rename(columns = diccionario, inplace = True)` cambia los nombres de las columnas según el diccionario

Eliminar columnas

`df.drop(columns = ["columna1", "columna2"], axis = b, inplace=True)` eliminar una o mas columnas o filas segun lo que especificamos

Reordenar columnas

`df = df.reindex(columns = lista_reordenada)` cambia el orden de las columnas del dataframe segun el orden de la lista reordenada

Crear columnas

Creacion de ratios

`df["columna?ratio"] = df.apply(lambda df: df["columna1"] / df["columna2"], axis = 1)`

Creacion de porcentajes

```
def porcentaje(columna1, columna2):
    return (columna1 * 100) / columna2

df["columna_%"] = df.apply(lambda df: porcentaje(df["columna1"], datos["columna2"]), axis = 1)
```

`df["nueva_columna"] = np.where(df["nombre_columna"] > n, "categoria_if_true", "categoria_if_false")` crea una nueva columna basada en una condición

`df["nueva_columna"] = np.select(lista_de_condiciones, lista_de_opciones)` crea una nueva columna con los valores basados en multiples condiciones
`df["columna_nueva"] = pd.cut(x = df["nombre_columna"], bins = [n,m,l..], labels = ['a', 'b', 'c'])` separa los elementos de un dataframe en diferentes intervalos (n-m, m-l, etc), creando una columna nueva que indica en cual intervalo cae el valor; con labels se puede asignar un string a cada intervalo

Crear columnas

`df["nueva_columna"] = (df["etiqueta_columna"] + x)` crea una nueva columna basada en otra
`df = df.assign(nueva_columna= df["etiqueta_columna] + x)` crea una nueva basada en otra
`df = df.assign(nueva_columna= [lista_valores])` crea una nueva columna de una lista de valores *tiene que ser de la misma longitud como el número de filas del dataframe*
`df.insert(indice_nueva_columna, "nombre_columna", valores)` crea una nueva columna en la indice indicada
`allow_duplicates = True` parametro cuando queremos permitir columnas duplicadas (por defecto es False)

Apply

apply() toma una función como argumento y la aplica a lo largo de un eje del DataFrame
`df['columna_nueva'] = df['col_1'].apply(función)`
crea una columna nueva con los valores de otra columna transformados según la función indicada
`df['columna_nueva'] = df['col_1'].apply(lambda x: x.método() if x > 1)`
crea una columna nueva con los valores de otra columna transformados según la lambda indicada
`df['columna_nueva'] = df.apply(lambda nombre: función(nombre['columna1'], nombre['columna2']), axis = b)` crea una columna nueva usando una función que coge dos parámetros (columna 1 y columna2)
`df.applymap(funcion, na_action=None, **kwargs)` acepta y devuelve un escalar a cada elemento de un dataframe; se tiene que aplicar a todo el DataFrame
`df['columna'] = df['columna'].map(mapa, na_action = 'ignore)` reemplaza valores de la columna según el mapa, que puede ser un diccionario o una serie; solo se puede aplicar a una columa en particular.

apply() con datetime

`df['columna_fecha'] = df['columna_fecha'] .apply(pd.to_datetime)` cambia una columna de datos tipo fecha en el formato datetime

```
def sacar_año(x):
    return x.strftime("%Y")
```

`df['columna_año'] = (df['columna_fecha'] .apply (sacar_año)` crea una columna nueva del año solo usando un método de la librería datetime; ("%B") para meses

Cambiar valores

Reemplazar valores basados en indices y condiciones:

```
indices_filtrados = df.index[df["columna"] == "valor"]
for indice in indices_filtrados:
    df["nombre_columna"].iloc[indice] = "valor_nuevo"
```

Reemplazar valores basados en metodos NumPy:

`df.replace(to_replace = valor, value = valor_nuevo, inplace = True)` reemplaza cierto valor por otro que especificamos
`df["nombre_columna"].replace(to_replace = valor, value = valor_nuevo, inplace = True)` reemplaza cierto valor en una columna por otro que especificamos
`df[["columna1", "columna2"]] = df[["columna1", "columna2"]].replace(r"string", "string", regex=True)` cambiar un patron/string por otro en multiples columnas
`df["nombre_columna"] = df["nombre_columna"] + x` reemplaza los valores de la columna por el valor + x (o otro valor que indicamos)

datetime

`import datetime`
`datetime.now()` o `datetime.today()` devuelve la fecha actual
`timedelta(n)` representa una duración la diferencia entre dos instancias; n es un numero de días pero se puede especificar days, seconds o microseconds
`ayer = datetime.now() - timedelta(1)`
`ayer = datetime.strftime(ayer, '%Y-%m-%d')`
`df["fecha"] = ayer` crea una columna con la fecha de ayer

`strftime()` nos permite crear formatos mas legibles de datos de tipo datetime
`datetime.strftime(variable_fecha, '%Y-%m-%d')` formatea la fecha al formato indicado

Sintaxis de strftime()

`%a` día de la semana abreviada (Sun)
`%A` día de la semana (Sunday)
`%w` día de la semana de 0 (domingo) a 6 (sábado)
`%b` mes abreviada (Sep)
`%B` mes (September)
`%m` mes con un zero (09)
`%-m` mes como float
`%d` día del mes con un zero (08)
`%-d` día del mes como float (8)
`%y` año sin siglo (99)
`%Y` año con siglo (1999)
`%Z` zona horaria (UTC)
`%p` AM o PM
`%c` fecha y hora
`%x` fecha
`%X` hora (07:06:05)
`%H` hora con zero (07)
`%-H` hora como float (7)
`%M` minuto con zero (06)
`%-M` minuto como float (6)
`%S` segundos con zero (05)
`%-S` segundos como float (5)