**Clase 12 – Data Wrangling**

**Data Wrangling:** Tarea de **convertir y mapear datos de un formato a** otro. Por ejemplo, se pueden convertir o mapear grandes cantidades de datos sin procesar a un formato diferente, de forma tal que sean más útiles para su consumo y análisis.

1. Primero se extraen los datos de una fuente en un **formato sin procesar (raw/crudos)**.
2. Luego se envían a un algoritmo, o se “**parsean**” en una **estructura de datos predefinida**.
3. Finalmente, se **almacenan para usarlos en el futuro**.

**Data Cleaning:** Proceso de **encontrar, corregir y/o eliminar registros incorrectos o inexactos** de un conjunto de registros o fuente de datos.

Puede incluir actividades tales como **eliminar errores tipográficos** o **validar y corregir errores** con una lista conocida de entidades. Incluye también la **estandarización de datos**. Ayuda a dar **consistencia de datos** a diferentes conjuntos provenientes de **varias fuentes de datos** que fueron **fusionados**.

Tanto **Data Wrangling** como **Data Cleansing** juegan un **papel importante** en el **preprocesamiento de datos** que hace falta para poder aplicar **algoritmos de aprendizaje automático** (**Machine Learning**) y **aprendizaje profundo** (**Deep Learning**)**.**

**Data Wrangling** es el **proceso de limpieza y unificación** de conjuntos de **datos** **desordenados y complejos** para facilitar su **acceso, exploración, análisis y modelización posterior**.

**Data Munging:** **arduo proceso de Limpiar, Preparar y Validar los Datos.**

**ETL** (**E**xtract, **T**ransform and **L**oad): Extraer, Transformar y Cargar los datos transformados.

**EDA** (**E**xploratory **D**ata **A**nalisys): **Resumir** las **principales características** del conjunto de **datos**, generalmente con **métodos visuales**. Puede o no usarse un **modelo estadístico,** la idea es ver **qué nos pueden decir** los datos, además de la tarea de modelado formal o prueba de hipótesis.

**Data Wrangling en Pandas**:

**Reemplazo de valores con pandas:**

Tenemos un DataFrame df. Podemos reemplazar todos los valores de 0 por 5 con el método **replace**:

**En Python:**

df.replace(0,5)

Otra posible aplicación es reemplazar todos los valores de la columna A de 0 y los valores de la columna B de 5 por 100:

df.replace({‘A’:0, ‘B’:5},100)

Otra posible aplicación es reemplazar 2 o más valores de una columna dada por un valor, ejemplo los valores de la columna A de 0 por 100 y de la columna A de 4 por 400:

df.replace({‘A’: {0:100, 4:400}})

Yendo a una aplicación aún más compleja, podemos 2 valores de una columna por 2 valores diferentes y un valor de otra columna por otro valor distinto:

df.replace({‘A’: {0:100,4:100},’C’:{‘e’:’z’}})

Con el método **duplicated** se puede definir en el argumento **subset** una lista de columnas a evaluar cuando queremos ver si dos registros son o no duplicados. Si no definimos subset, entonces el método considerará duplicados sólo a aquellos registros en los que todas las columnas del dataset tengan los mismos valores. El argumento **keep** puede tomar distintos valores: Con first, devuelve como False el primer registro encontrado y como true los subsiguientes; con last devuelve como False el último registro encontrado y como true los anteriores; con False devuelve como True todos los registros duplicados.

Con el método **drop\_duplicates** se puede obtener un nuevo DataFrame sin registros duplicados. También tiene los argumentos **subset** y **keep** y funcionan igual que con el método duplicated.

**En Python:**

# Dado un dataframe data, generamos un DataFrame data\_state con las columnas “state”, “StateName” y “Region”:

data\_state = data.loc[:,[‘state’,’StateName’,’Region’]]

# Vamos a estudiar si hay duplicados en data\_state:

data\_state\_duplicates\_mask = data\_state.duplicated(keep = ‘first’)

print(‘Registros Duplicados en Data\_State: ‘, any(data\_state\_duplicates\_mask) )

print(‘Cantidad de Registros Duplicados en Data\_State: ‘,data\_state\_duplicates\_mask.sum())

# Ahora vamos a tomar otro approach y estudiar si hay duplicados en data:

data\_ duplicates\_statefields= data\_state.duplicated(subset = [‘state’,’StateName’, ‘Region’], keep = ‘first’)

print(‘Registros Duplicados en Data (considerando datos de estado): ‘, any(data\_ duplicates\_statefields) )

print(‘Cantidad de Registros Duplicados en data (Considerando datos de Estado): ‘,data\_ duplicates\_statefields.sum())

# Por último dropeamos los valores duplicados:

data\_state\_nodup = data\_state.drop\_duplicates(keep = ‘first’)

# No dropea los duplicados en data porque tenemos distintos registros con la producción de distintos años por estado, al no contemplar el año, perderíamos muchísima información que no queremos perder.

Con el método **rename** podemos cambiar las etiquetas de los ejes índices y de las columnas. Se puede usar el argumento **mapper** para dar como input un diccionario con claves como etiquetas a reemplazar y valores como etiquetas de reemplazo; con el argumento **axis** se puede indicar si el reemplazo será sobre el índice o sobre las columnas. El argumento **index** es un equivalente a usar el argumento mapper con axis = 0. El argumento **columns** es un equivalente a usar el argumento mapper con axis = 1.

**En Python:**

# Vamos a renombrar dos columnas con los argumentos mapper y axis:

mapper ={‘nIMIDACLOPRID’:’ nimidacloprid‘, ‘nTHIAMETHOXAM’: ‘nthiamethoxam’}

data\_renamed\_cols = data.rename(mapper, axis = 1)

# Otra forma:

data\_renamed\_cols2 = data.rename(columns = mapper)

# Solo se modifican aquellas columnas especificadas en el diccionario, sin necesidad de incluir las que queremos mantener como están.

**Variables Categóricas**

Las variables se pueden clasificar como:

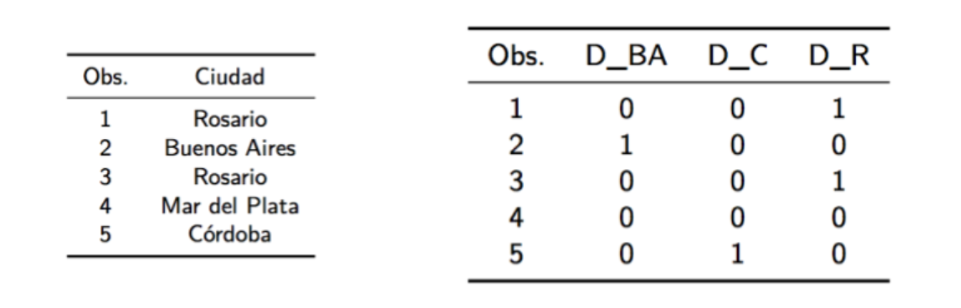
* **Cuantitativas:** Tienen valores numéricos, tales como el precio de una casa o el salario de una persona.
* **Cualitativas:** Tienen valores en una de K diferentes categorías. En el caso puntual en que K = 2 (o sea sólo 2 posibles valores) se la llama **binaria** o **dicotómica**.

A su vez, las **variables cualitativas** pueden clasificarse como:

* **Nominal/Categórica. Categorías Nombradas**: Suelen asignarse **valores** o **rótulos numéricos**. Ejemplo: Estado civil: 0 – soltero; 1 – casado, 2 – divorciado. Dichos números son arbitrarios; como el software asume que representan cantidades algebráicas, se los asocia con un cierto orden. La principal medida de posición en estos casos es la **moda**, ya que no tienen ni media ni mediana, al no ser numéricos.
* **Ordinal**: Simil categórica, pero con un orden claro: Ejemplos: desde 1- muy insatisfecho hasta 5 – muy satisfecho.
* **Dummy**: Variable cualitativa que toma 0 o 1 para indicar la ausencia o presencia de algún atributo o efecto categórico (simil distribución Bernoulli). Se puede expresar con una función indicadora. Una variable categórica con N variables puede expresarse en términos de N-1 variables Dummies (one-hot encoding). Si las categorías tienen muchos valores, aumenta considerablemente la dimensionalidad de los datos.

IE:

Dada una variable categórica C, que registra la ciudad en la que vive una muestra de habitantes de Argentina, con 4 valores posibles: Buenos Aires, Rosario, Córdoba y Mar del Plata. Suponiendo que tenemos 5 observaciones, es posible representar estas observaciones usando dummies:



**Si existen k categorías**, con **k-1 variables dummies alcanza para representarlas.**

**Data Wrangling en Pandas (reloaded)**

**Discretización:** Proceso de **Transformar una variable numérica en categórica**.

Con el método **cut** se pueden **separar valores en intervalos discretos**, devolviendo el intervalo semi-cerrado que le corresponde a cada valor. De esta forma es posible **transformar variables continuas en categóricas.**

Con el argumento **bins** se puede con un entero indicar la **cantidad de intervalos de igual “ancho”** a construir; **o** bien **podemos con una lista especificar los límites** de cada categoría.

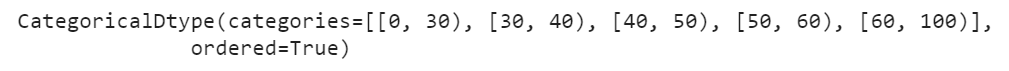
Con el argumento **right** le indicamos si queremos que el/los intervalos/s incluya/n el valor límite por la derecha.

IE: **En Python**

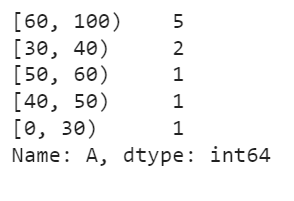
Tenemos un DataFrame con 2 columnas de valores aleatorios entre 0 y 100 y entre 100 y 500, con 10 registros cada una. Con el método **.describe()** podemos sacar las principales características de cada columna (cuenta de registros, media, desvío estándar, mínimo; 25%, 50%, 75%, máximo). Podemos definir 5 categorías para la categoría A con límites dados a partir de una lista ejemplo:

bins = [0,30,40,50,60,100]

A\_categories = pd.cut(Data.A, bins, right = False)



A\_categories.value\_counts()

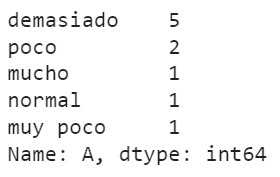


Con el argumento **labels** es posible crear categorías con etiquetas asociadas:

group\_labels = [‘muy poco’, ‘poco’, ‘normal’, ‘mucho’, ‘demasiado’]

A\_categories\_labels = pd.cut(data.A, bins, labels = group\_labels)

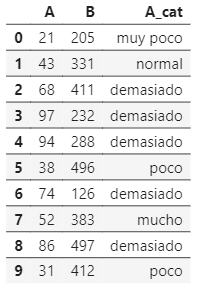
A\_categories\_labels.value\_counts()



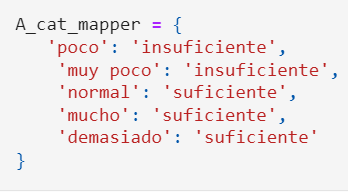
Podemos agregar una nueva columna a un DataFrame en Pandas usando un diccionario si las claves del mismo tienen relación con la serie de valores de alguna columna del DataFrame, agregando en la nueva columna los valores del diccionario.

Ejemplo: **En Python**

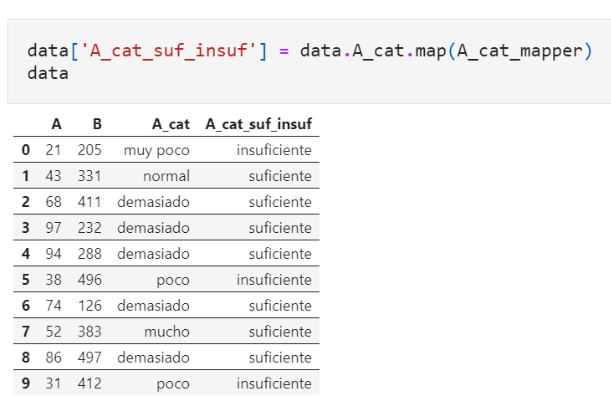
Dado el DataFrame data:



Supongamos que queremos cambiar los valores de A\_cat. Creamos un diccionario que tenga como claves los valores de A\_cat y como valor los valores deseados:



Con el método **.map(nombrediccionario)** podemos agregar la nueva columna:

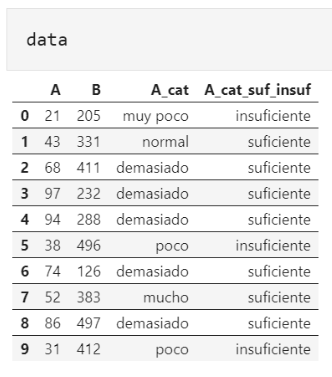


A diferencia del método .**replace()** el método **map()** aplica **sólo sobre una columna del DataFrame**.

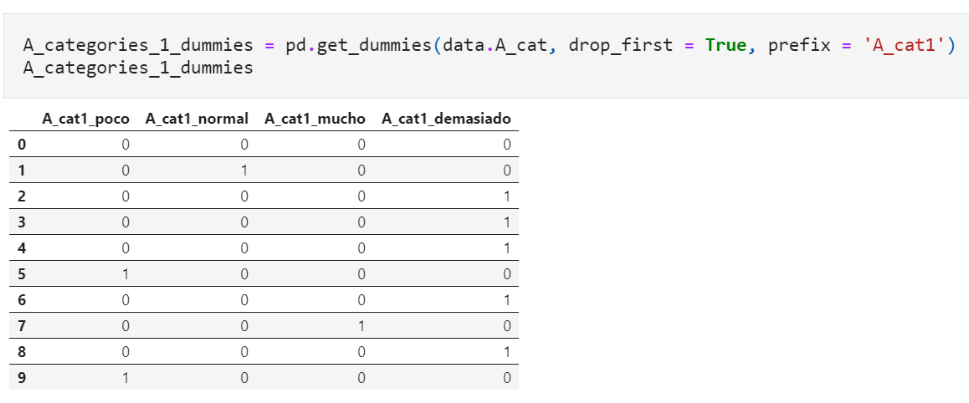
Usando el método **pd.get\_dummies** es posible recibir una serie o lista de series y realizar un hot encoding. Una variable con k categorías es representable con k-1 variables. Con el parámetro **drop\_first = True** se generan k-1 variables. Con el argumento **prefix** podemos establecer un prefijo para el nombre de cada una de las categorías que representan la variable. Esto es particularmente útil cuando construimos dummies de más de una columna de un DataFrame, para evitar confundirnos sobre cuál es la variable original que corresponde a cada columna de categoría.

Ejemplo: **En Python**

Tenemos el siguiente DataFrame:



Usamos el método get\_dummies sobre la columna A\_cat para generar un nuevo DataFrame con k-1 categorías (en este caso 4 categorías en vez de 5, representadas en 4 columnas):



Ahora lo hacemos trabajando en la columna A\_cat\_suf\_insuf, que tiene sólo 2 valores posibles; lo va a mantener en una columna, pero con valores 1 o 0:

A\_categories\_2\_dummies = pd.get\_dummies(data.A\_cat\_suf\_insuf, drop\_first = True, prefix = ‘A\_cat2’)

**Timestamp y Period**

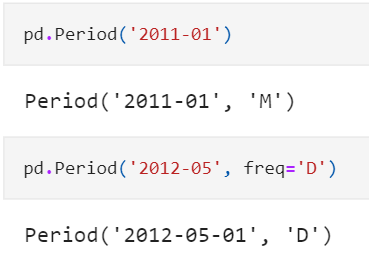
**Unix TimeStamp** es una manera de **trackear el tiempo**. Se considera en segundos desde el 1 de Enero de 1970 UTC 0:00 hs hasta la fecha. Al tener un punto de referencia fijo, es muy simple manejar tiempo y fechas en distintos sistemas y arquitecturas. Hay páginas que ingresando el TimeStamp devuelven la fecha y hora desglosada en años, meses, días, horas, minutos y segundos. Ejemplo: <https://www.unixtimestamp.com/index.php>

Los **datos Timestamp** son la forma más básica de **Series de Tiempo**. Asocian valores con puntos en el tiempo. **Es posible instanciar objetos** de clase **Timestamp** en Pandas.

Podemos generar objetos Timestamp con el método **pd.Timestamp** a partir de una **Datetime**, un **string** o bien con una **tupla**:

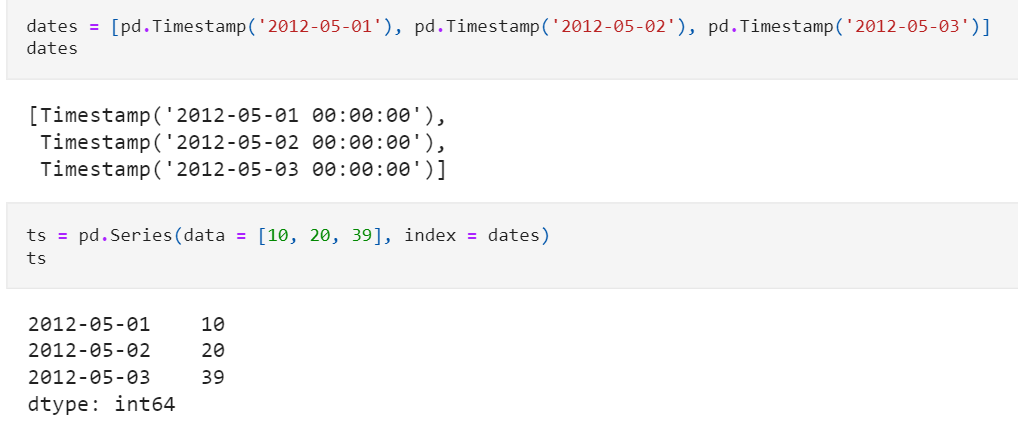


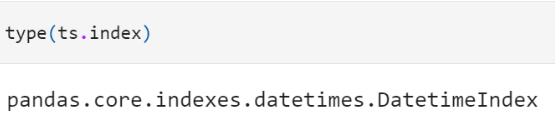
Como en muchos casos es más natural asociar variables a **intervalos de tiempo**, podemos representar un intervalo de tiempo con un objeto de clase **Period**, el cual puede ser instanciado explícitamente o inferido a partir de un string con cierto formato. Para todo esto nos sirve el método **pd.period**



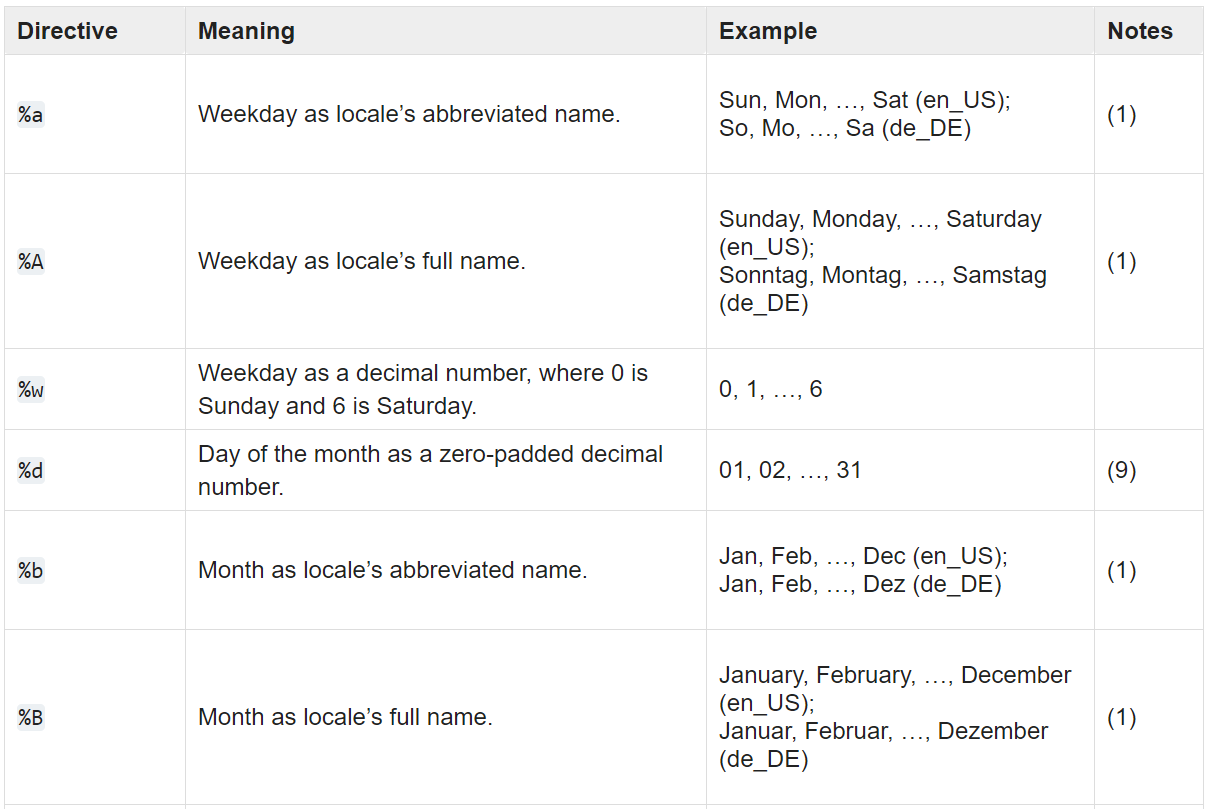
Es posible usar los objetos **Timestamp** y **Period** como **índices**. Listas que contengan estos objetos como índices convierten dichos índices automáticamente en objetos de tipo **DatetimeIndex** y **PeriodIndex** respectivamente.

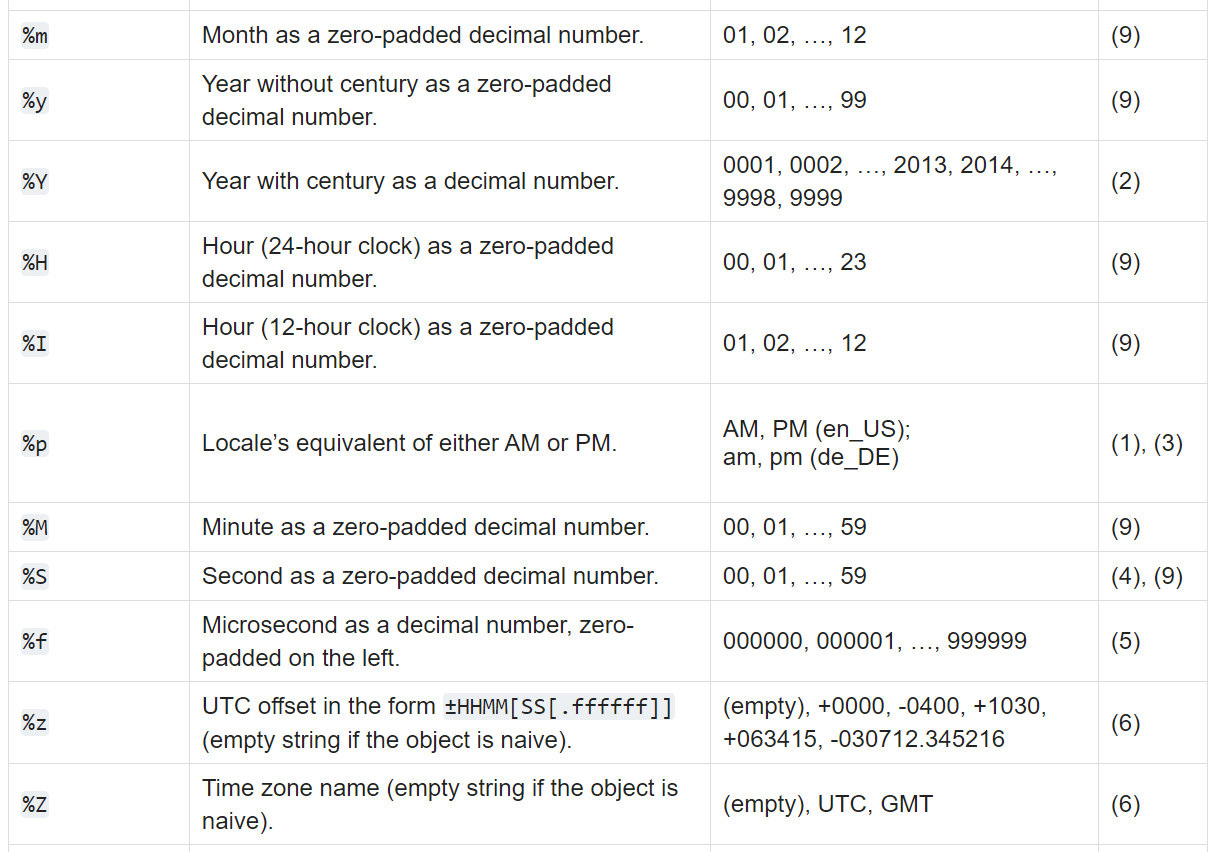
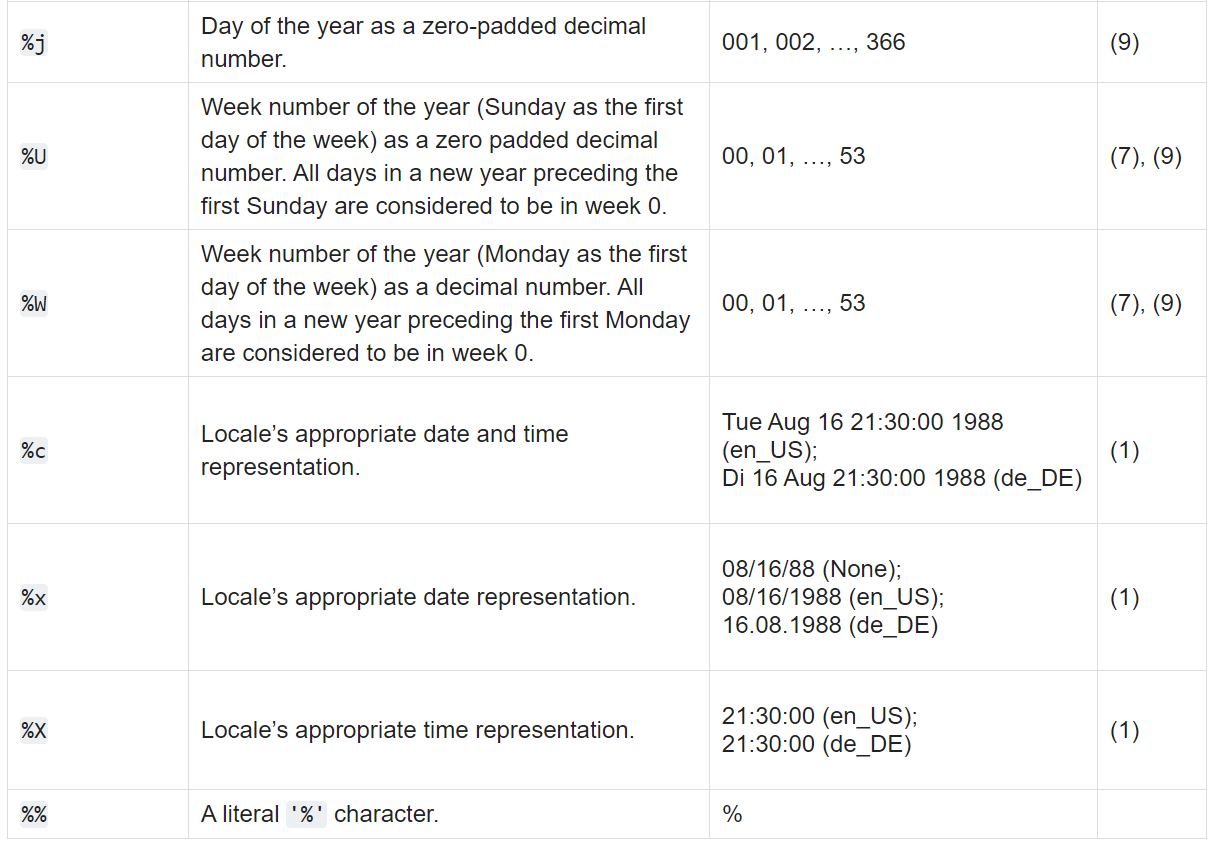
Ejemplo: Tenemos una lista de fechas Timestamp. Generamos una serie con esta lista como índice. Al consultar el tipo de objetos del índice de la serie recién generada, nos indica que es un DatetimeIndex:





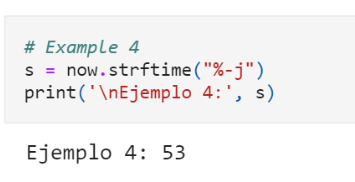
Con el módulo **datetime** podemos acceder a los tipos de dato de fecha y hora en Python, con numerosas funciones para trabajar con fecha y hora. Con el método **strftime()** es posible convertir objetos de fecha y hora en su representación como cadena de caracteres. Tiene un estilo análogo al uso de metacaracteres.



Ejemplos varios: **En Python**





Con el método **.to\_datetime** podemos convertir un argumento en un objeto de tipo datetime indicando en el argumento **format** cómo le estamos dando la información. Con el argumento **unit** podemos especificar si queremos el output en segundos o nanosegundos (seguramente haya más opciones). Por default da formato fecha, y , si dispone de la información, fecha, hora, minutos, segundos.

Ejemplos varios: **En Python**

