Modificar y limpiar datos utilizando pandas

Contents


```
import os
# Por precaución, cambiamos el directorio activo de Python a aquel que contenga este notebook
if "PAD-book" in os.listdir():
    os.chdir(r"PAD-book/Laboratorio-Computacional-de-Analytics/S5 - Extraer, transformar y cargar datos/S5.TU1/")
```

Con lo aprendido hasta ahora, somo capaces de cargar un `DataFrame` que contenga los datos a analizar. Además, podemos indexar este `DataFrame` en las filas según el contexto del problema y renombrar sus columnas según las prácticas recomendadas. No obstante, antes de realizar un análisis de datos, debemos revisar algunos aspectos importantes del manejo de bases de datos. Aprenderemos acerca de las estrategias que podemos aplicar sobre registros faltantes, y sobre el manejo de cadenas de texto y su transformación a fechas en los `DataFrame`.

Requisitos

Para desarrollar este tutorial necesitarás:

- Conocer las principales características de los arreglos en `numpy` y de los `DataFrame` de `pandas`.
- Importar datos a un `DataFrame` desde distintos formatos.
- Indexar un `DataFrame` a partir de una o más columnas.

Objetivos

Al final de este tutorial podrás:

1. Modificar cadenas de texto contenidas en un `Series`.
2. Representar fechas utilizando el paquete `datetime`.
3. Imputar datos faltantes en un `DataFrame`.

1. Modificar cadenas de texto contenidas en un `Series` utilizando la librería `pandas`

En `pandas`, las cadenas de texto contenidas en un `Series` (o en columnas de un `DataFrame`) heredan varios métodos propios de los objetos de tipo cadena de texto. Para acceder a ellos los llamamos del módulo `Series.str` (o `DataFrame.str`).

A continuación explicamos la documentación de algunos de los métodos más comunes: `split`, `get_dummies`, `capitalize`, `lower`, `upper` y `strip`.

- `split`: divide cada cadena del `Series` cada vez que encuentra la subcadena especificada en el parámetro `pat`. Utilizamos el parámetro `n` para limitar la cantidad de divisiones que realizamos las cadenas. Utilizamos el parámetro expand para retornar los segmentos de las cadenas en un `DataFrame` (`expand = True`) o en un `Series` de listas (`expand = false`).

- `get_dummies`: crea un `DataFrame` donde las columnas corresponden a cada uno de los valores únicos del `Series`, las filas corresponden a los elementos del `Series` y los valores son binarios (`0` o `1`) dependiendo de si el elemento toma el valor que indica la columna.

- `capitalize`: revisa si el primer caracter de cada cadena es una letra minúscula y lo convierte en mayúscula y vuelve minúscula el resto de las letras en la cadena.

- `lower`: convierte todas las letras de cada cadena en minúsculas.

- `upper`: convierte todas las letras de cada cadena en mayúsculas.

- `strip`: elimina del inicio y del final de cada cadena del `Series`, la subcadena especificada en el parámetro `to_strip`. Si la subcadena no es especificada, por defecto se supone la subcadena `" "` (espacio).

Importamos el paquete `pandas`.

```
import pandas as pd
```

Ejemplo 1

En la siguiente celda de código declaramos un `Series` con el nombre de algunos libros. Se nos pide que extraigamos el artículo del título de los libros que se encuentran dentro del `DataFrame`.

```
serie_libros = pd.Series(["El Extranjero", "La Peste", "La Caída"])
serie_articulos = serie_libros.str[:2] # Tomamos los primeros dos caracteres del
título.
serie_articulos
```

```
0    El
1    La
2    La
dtype: object
```

Puede cumplirse lo mismo con el método `split`, como se ve a continuación:

```
serie_articulos = serie_libros.str.split(' ', expand=True)
serie_articulos[0]
```

```
0    El
1    La
2    La
Name: 0, dtype: object
```

2. Representar fechas utilizando la librería `datetime`

Una de las grandes ventajas de la programación de rutinas de código en Python es la versatilidad que ofrece. El caso de la manipulación de información temporal no es la excepción, en tanto Python permite representar los principales formatos de fechas que se usan en lenguajes de manipulación de bases de datos como SQL. El principal módulo a importar para trabajar con fechas es `datetime`.

2.1. El módulo `datetime`

La principal característica del módulo `datetime` es que permite trabajar simultáneamente con fechas (*Dates*) y tiempos (*Time*). Es decir, al crear un objeto con el módulo `datetime`, este contendrá la información de la fecha (día, mes y año) y un registro detallado del dato de tiempo (hora, minuto, segundo, microsegundo). Para importar el módulo `datetime` se puede usar la siguiente sintaxis:

```
from datetime import datetime
```

Esta sintaxis permite declarar de forma directa objetos de tipo `datetime` ya que importa los métodos del módulo (y no todo el paquete) `datetime`. Así, resulta innecesario referenciar nuevamente un objeto de este módulo. Para declarar una fecha se usa la siguiente sintaxis, empleando valores enteros como argumentos:

```
fecha = datetime(anho, mes, dia)
```

A esta fecha se le asignará por defecto la primera hora del día (00:00:00). Si se desea, también se puede especificar el total de la información como en la siguiente declaración:

```
tiempo = datetime(anho, mes, dia, hora, minuto, segundo, microsegundo)
```

A continuación, trabajaremos los principales métodos del módulo `datetime`.

2.2. Métodos de los objetos tipo `datetime`

Métodos	Descripción
`weekday()`	Obtiene el día de la semana correspondiente a la fecha
`isoweekday()`	Obtiene el día de la semana correspondiente a la fecha en formato ISO
`__format__(formato)`	Confiere a un `datetime` el `formato` especificado
`replace()`	Permite modificar cualquiera de los valores de una fecha particular (año, mes, dia)

2.3 Métodos del módulo `datetime`

Métodos	Descripción
`stftime(formato)`	Retorna una cadena de texto con la información de la fecha según el `formato` especificado
`strptime(cadena_fecha)`	Crea un `datetime` con base en una fecha descrita por una cadena de texto (`cadena_fecha`)

Para usar los métodos `stftime` y `strptime` se usan las siguientes convenciones:

Convención	Descripción
`'%a'`	Referencia los primeros caracteres del día de la semana 'Wed'
`'%A'`	Referencia el nombre completo 'Wednesday'
`'%B'`	Referencia el nombre completo del mes 'Septiembre'
`'%w'`	Referencia el día de la semana con números del 0 al 6 donde el Domingo es el 0
`'%m'`	Referencia el número del mes del '01' al '12'
`'%p'`	Referencia la hora en formato AM/PM
`'%y'`	Referencia el año usando únicamente los últimos dos dígitos
`'%Y'`	Referencia el año usando todos los dígitos
`'%Z'`	Referencia la zona horaria
`'%z'`	Referencia la zona horaria en formato UTC
`'%j'`	Referencia el día del año del '001' al '366'
`'%W'`	Referencia el día
`'%U'`	Referencia el número de la semana en el año desde el '00' hasta el '53'

Ejemplificaremos el uso de los métodos `strftime` y `strptime` según estas.

Importamos el módulo `datetime`.

```
from datetime import datetime
```

Ejemplo 2

Vamos a convertir la cadena de texto con formato `"%y-%m-%d"` en un objeto `datetime`.

```
cadena_fecha = "18-12-31"
fecha = datetime.strptime(cadena_fecha,"%y-%m-%d")
fecha
```

```
datetime.datetime(2018, 12, 31, 0, 0)
```

También se habría podido lograr esta fecha con la cadena de texto que contiene todos los dígitos del año:

```
cadena_fecha = "2018-12-31"
fecha = datetime.strptime(cadena_fecha,"%Y-%m-%d")
fecha
```

```
datetime.datetime(2018, 12, 31, 0, 0)
```

Recuerde que también se puede recuperar la cadena de texto original sobre la cual se creó la fecha usando el método `strftime`.

```
cadena_fecha = datetime.strftime(fecha,"%Y-%m-%d")
cadena_fecha # tipo: cadena de texto
```

```
'2018-12-31'
```

2.3. Columnas con formato de fechas

Una vez conocemos el procedimiento para transformar cadenas de texto en objetos `datetime` podemos pensar en cómo transformar columnas o `Series` de enteros o cadenas de texto en fechas. La manera más frecuente es por medio del método `to_datetime` de `pandas`:

```
to_datetime(arg=cadena_fecha, UTC=None, format=None)
```

A continuación, describimos el uso de los parámetros.

- `arg`: cadena de texto, entero, lista, arreglo o fecha (`datetime`) a transformar en una fecha (`datetime`).
- `UTC`: especifica la zona horaria.
- `format`: recibe una cadena de texto que indica el formato de la fecha según las convenciones.

Ejemplo 3

En la celda de código a continuación, importamos el índice de los títulos de deuda pública de los TES de Corto Plazo para los años comprendidos entre el 2010 y 2019. Estos datos fueron descargados directamente de la página de la Bolsa de Valores de Colombia disponible en las referencias. Declaramos un `DataFrame` indexado por su columna `"fecha"`.

Empecemos inspeccionando la base de datos:

```
dfTES = pd.read_excel('Archivos/Indices.xls', index_col=0, header=1)
dfTES.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 2902 entries, nan to nan
Data columns (total 8 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   Fecha                2902 non-null   int64  
 1   Indice               2902 non-null   object 
 2   Valor Hoy            2902 non-null   float64
 3   Valor Ayer           2902 non-null   float64
 4   Variacion %          2902 non-null   float64
 5   VariaciÃ³n Absoluta  2902 non-null   float64
 6   Variacion 12 meses   2902 non-null   float64
 7   VariaciÃ³n Anual     2902 non-null   float64
dtypes: float64(6), int64(1), object(1)
memory usage: 204.0+ KB
```

Notamos que la base de datos está indexada por el número de fila, por lo que podemos indexar el `DataFrame` en la fecha sin perder información, como se muestra a continuación:

```
dfTES.index = pd.to_datetime(dfTES["Fecha"], format="%Y%m%d")
dfTES.index
```

```
DatetimeIndex(['2010-01-01', '2010-01-02', '2010-01-03', '2010-01-04',
               '2010-01-05', '2010-01-06', '2010-01-07', '2010-01-08',
               '2010-01-09', '2010-01-10',
               ...
               '2019-12-16', '2019-12-17', '2019-12-18', '2019-12-19',
               '2019-12-20', '2019-12-23', '2019-12-24', '2019-12-26',
               '2019-12-27', '2019-12-30'],
              dtype='datetime64[ns]', name='Fecha', length=2902, freq=None)
```

Como el índice es de tipo `datetime`, podemos utilizar los métodos de este módulo. Por ejemplo, `weekday` para el día de la semana.

```
dfTES.index.weekday
```

```
Int64Index([4, 5, 6, 0, 1, 2, 3, 4, 5, 6,
            ...
            0, 1, 2, 3, 4, 0, 1, 3, 4, 0],
           dtype='int64', name='Fecha', length=2902)
```

3. Imputar datos faltantes en un `DataFrame`, utilizando la librería `pandas`

3.1. Representación de datos faltantes dentro de Python

Python cuenta con dos alternativas para representar datos faltantes: el objeto `None` y el valor `nan` (*Not a Number*) del paquete `numpy`. El valor `nan` es almacenado como un objeto de tipo `float`. Así, podemos aplicar métodos de `numpy` sobre arreglos numéricos, incluso si contienen valores `nan`. Dado que `nan` es un elemento del paquete, podemos asignarlo a variables o invocarlo de la misma manera que cualquier método o constante.

Importemos el paquete `numpy`.

```
import numpy as np
```

Al aplicar métodos de `numpy` sobre arreglos que contienen valores `nan`, el resultado suele ser `nan`.

```
numeros = np.array([1, 2, 3, 4, np.nan])
promedio = np.mean(numeros)
promedio
```

```
nan
```

Pese a no ser un número, `nan` es de tipo numérico (`float`).

```
type(np.nan)
```

```
float
```

En este caso, para poder calcular el promedio utilizamos el método `nanmean`, el cual omite los valores `nan`.

```
numeros = np.array([1, 2, 3, 4, np.nan])
promedio = np.nanmean(numeros)
promedio
```

```
2.5
```

Ninguno de estos métodos sirve para un objeto `None`, ya que `numpy` no lo tiene definido como de tipo numérico. Veamos un ejemplo.

```
numeros = np.array([1, 2, 3, 4, None])
try:
    np.mean(numeros)
except:
    print("El arreglo tiene valores faltantes.")
try:
    np.nanmean(numeros)
except:
    print("El arreglo contiene valores no numéricos.")
```

```
El arreglo tiene valores faltantes.
El arreglo contiene valores no numéricos.
```

3.2. Imputar datos faltantes utilizando la librería `pandas`

Debido a que `numpy` puede operar sobre los objetos `nan` y no sobre los `None`, `pandas` representa los datos faltantes en un `Series` con el valor `nan`. Al introducir valores faltantes dentro de un `Series`, `pandas` transforma el tipo de dato para que sea compatible con el de los valores `nan`.

Al introducir un valor `nan` a un `Series`, `pandas`:

- convierte el `Series` en tipo `float`, si este es un `Series` de enteros.
- convierte el `Series` en tipo `object`, si este es un `Series` de valores lógicos.
- conserva el tipo del `Series`, si este es de tipo `float` u `object`.

Nota: recuerda que toda la información aplicable a un `Series` es igualmente aplicable a una columna de un `DataFrame`.

Ejemplo 4

Para ilustrar el cambio de tipo de dato, creamos un `Series` con los enteros del 1 al 10. Ya creado, reemplazamos el primer elemento por un `nan`.

```
numeros = pd.Series(range(11), dtype=int)
numeros[0] = np.nan
numeros
```

```
0      NaN
1      1.0
2      2.0
3      3.0
4      4.0
5      5.0
6      6.0
7      7.0
8      8.0
9      9.0
10    10.0
dtype: float64
```

Confirmamos que el `Series` en la variable `numeros` pasó de ser de tipo entero a ser de tipo `float`. Ahora bien, ¿cómo haríamos para detectar los faltantes y después eliminarlos o reemplazarlos?. Hay varias maneras, una de ellas es usar los métodos `notnull` e `isnull`. Estos métodos generan un `Series` de booleanos que identifican los datos faltantes con el valor de `True` para el método `isnull` y `False` para el método `notnull`. El siguiente ejemplo ilustra el uso del método `notnull`.

Ejemplo 5

Vamos a usar el método `notnull` para eliminar los datos faltantes del `Series` en la celda de código:

```
numeros = pd.Series([*range(5), np.nan, *range(6,10), np.nan])
numeros
```

```
0     0.0
1     1.0
2     2.0
3     3.0
4     4.0
5     NaN
6     6.0
7     7.0
8     8.0
9     9.0
10    NaN
dtype: float64
```

```
numeros = numeros[numeros.notnull()]
numeros # Se quitaron las filas con índices 5 y 10
```

```
0    0.0
1    1.0
2    2.0
3    3.0
4    4.0
6    6.0
7    7.0
8    8.0
9    9.0
dtype: float64
```

3.3. Métodos para imputar faltantes utilizando la librería `pandas`

Existen dos métodos en `pandas` para manipular los datos faltantes: `dropna` y `fillna`.

Método `dropna`

Este método elimina aquellas columnas o filas que contengan entradas `nan`.

```
DataFrame.dropna(axis=0, how='any', thresh=0, subset=DataFrame.columns)
```

- `axis`: indica si aplicar el método sobre sobre las filas (`axis = 0`) o sobre las columnas (`axis = 1`). Por defecto es sobre las filas.

- `how`:
  - `how = 'any'`: elimina la fila o la columna si contiene al menos un faltante.
  - `how = 'all'`: elimina la fila o la columna si solo contiene faltantes.

- `thresh`: elimina todas las filas o columnas con más datos faltantes que el umbral (de tipo `int`) especificado.

- `subset`: permite seleccionar un subconjunto de columnas o de filas sobre el cual aplicar el método.

Ejemplo 6

A continuación, declaramos un `DataFrame` para ejemplificar el uso del método `dropna`.

```
tabla_numeros = pd.DataFrame([[1     , np.nan, np.nan, 2     ],
                              [np.nan, np.nan, 1     , np.nan],
                              [np.nan, 0     , np.nan, 2     ]],
                             columns=["A", "B", "C", "D"])
tabla_numeros
```

	A	B	C	D
0	1.0	NaN	NaN	2.0
1	NaN	NaN	1.0	NaN
2	NaN	0.0	NaN	2.0

Utilizamos el método `dropna` para eliminar aquellas filas que solo contienen datos faltantes en el subconjunto de columnas `A` y `C`.

```
tabla_numeros.dropna(how="all", subset=["A","C"])
```

	A	B	C	D
0	1.0	NaN	NaN	2.0
1	NaN	NaN	1.0	NaN

Método `fillna`

```
DataFrame.fillna(value=None, method=None, axis=None, inplace=False, limit=0)
```

- `value`: indica el valor o diccionario de valores para imputar en las entradas `nan`.
- `method`:
  - `method = 'ffil'`: (*forward fill*) rellena cada dato faltante con el dato no faltante anterior.
  - `method = 'bfill'`: (*backward fill*) rellena cada dato faltante con el dato no faltante siguiente.
- `axis`:
  - `0`: aplica el método sobre las filas.
  - `1`: aplica el método sobre las columnas.
- `inplace`:
  - `inplace = True`: aplica los cambios sobre la variable que invoca el método.
  - `inplace = False`: aplica los cambios sobre una copia de la variable que invoca el método.
- `limit`: limita el número máximo de datos a imputar hacia adelante (o hacia atrás), según lo especificado en el parámetro `method`.

Ejemplo 7

Vamos a completar los datos del siguiente `DataFrame` reemplazando los valores faltantes por su anterior no faltante en la misma fila.

```
tabla_numeros = pd.DataFrame([[1     , np.nan, np.nan, 2     ],
                              [np.nan, np.nan, 1     , np.nan],
                              [np.nan, 0     , np.nan, 2     ]],
                             columns  =["A", "B", "C", "D"])
tabla_numeros
```

	A	B	C	D
0	1.0	NaN	NaN	2.0
1	NaN	NaN	1.0	NaN
2	NaN	0.0	NaN	2.0

Para esto, debemos especificar `axis=1`. Además, como utilizaremos los valores precedentes, especificamos `method = ffill`.

```
tabla_numeros = tabla_numeros.fillna(axis=1, method="ffill")
tabla_numeros
```

	A	B	C	D
0	1.0	1.0	1.0	2.0
1	NaN	NaN	1.0	1.0
2	NaN	0.0	0.0	2.0

Ejemplo 8

Imputamos cada una de las entradas faltantes (columna `A`, fila `1`; columna `A`, fila `2`; columna `B` , fila `1`) del objeto `tabla_numeros` con el valor del promedio de la columna a la que pertenece.

```
tabla_numeros = tabla_numeros.fillna(tabla_numeros.mean())
tabla_numeros
```

	A	B	C	D
0	1.0	1.0	1.0	2.0
1	1.0	0.5	1.0	1.0
2	1.0	0.0	0.0	2.0

Referencias

Bolsa de Valores de Colombia (2020). Índices de TES de Corto Plazo [Base deDatos]. Recuperado el 14 de diciembre de 2020 de : https://www.bvc.com.co/pps/tibco/portalbvc

J. VanderPlas (2016) Python Data Science Handbook: Essential Tools for Working with Data O'Reilly Media, Inc.

Créditos

Autores: Jorge Esteban Camargo Forero, Alejandro Mantilla Redondo, Diego Alejandro Cely Gomez

Fecha última actualización: 15/07/2022