**Clase 9 – Limpieza de Datos**

**Limpieza y Preparación de Datos**:

**Data Munging** – Arduo proceso de limpiar, preparar y validar los datos.

Se gasta el 80% en limpiar y preparar la información.

**Limpieza de Datos con Pandas**

La limpieza es un paso necesario en todo proyecto de datos. Se puede resumir en 5 tareas:

* **Resolver problemas de formato y asignar los tipos correctos de datos**. Ejemplo, paso de CSV a Pandas y una fecha no se importa correctamente. En función del tipo de datos, algunas operaciones se pueden realizar o no. Además, algunos tipos de datos ocupan menos espacio en la memoria que otros.
* **Estandarizar categorías** Si se recolectó información en un sistema que no tiene los valores tipificados; puede que valores que representan la categoría estén expresados de forma distinta. Ej: Arg, AR y Argentina.
* **Corregir valores erróneos** IE: Valor numérico o inválido para describir género; o una edad con valor negativo o mayor que 100.
* **Completar datos faltantes**: Qué hacemos con la información que se perdió o que nunca se recolectó. Hay técnicas para completar los datos faltantes; al proceso de completar datos faltantes se lo llama **imputación**.
* **Organizar correctamente el Dataset:** Es importante estructurar las filas y columnas de la forma más conveniente. Para esto, se usan las reglas del **“tidy data”**.

**Datos Faltantes**: Los datasets del mundo real siempre tienen **datos faltantes**. Cada lenguaje lidia con ellos. En el caso de **Pandas**, por estar basado en **Numpy**, usa estos 3:

* **None:** Objeto de Python que representa **ausencia de dato**.
* **NaN:** Definición de valor **faltante de floats**.
* **NaT:** Valores faltantes de tipo **Timestamp** (fechas).

Es importante poder **detectar**, **rellenar o eliminar** los datos faltantes. Por eso, hay que tener conocimiento del dominio para poder **definir cuáles datos faltantes se completarán y cómo**.

Para identificar datos faltantes, es posible usar el método isnull; el cual devuelve otro DataFrame donde cada elemento es True o False según si ese elemento en el DataFrame original es o no null. Su método inverso es notnull (da True a los valores no nulos y False a os valores nulos).

IE: Si tenemos un DataFrame data, is\_null\_result = data.isnull()

Para ver cuántos nulos hay en cada columna podemos usar sum en el DataFrame resultante de usar el isnull:

cant\_nulls = is\_null\_result.sum()

En caso de querer ver cuántos nulos hay en cada fila, tendríamos que usar:

cant\_nulls = is\_null\_result.sum(axis = 1)

La Imputación puede ser **simple** o **múltiple:**

* **Imputación Simple:** asignar un valor por cada dato faltante a partir de la propia variable o de las demás variables para completar la base de datos.
* **Imputación Múltiple:** A cada valor faltante se le asigna un grupo de **m** valores, generando **m** bases de datos completas. En cada una de estas nuevas bases de datos se estiman los parámetros de interés y luego se combinan los resultados obtenidos.

Pandas ofrece varias formas de hacer esto:

1. **Analisis con datos completos**: Eliminar los registros que presenten algún dato faltante. Hay varios approachs posibles.

**Ventaja**:Fácil de Implementar. **Desventaja:** Pérdida importante de información; posibles sesgos en los estimadores de los parámetros. El método asume que la falta de respuesta se dio en forma aleatoria, cuando en general no fue así. Método ideal sólo en aquellos casos en los que los datos que faltan sean pocos y al azar.

Dado el DataFrame Data, usa la sintaxis:

data.dropna(axis=0, how = ‘any’, inplace = True)

Con esto baja el shape del dataframe de (13580, 21) a (6196, 21).

**Eliminar todas las columnas que tengan algún valor faltante:**

data\_columnas\_completas = data.dropna(**axis = 1**)

**Eliminar todas las filas que tengan algún valor faltante:**

data\_filas\_completas = data.dropna() o bien

data\_columnas\_completas = data.dropna(**axis = 0**)

**Importante:** Si definimos el argumento **inplace = True**, entonces dropna no sólo va a devolver un nuevo DataFrame, sino que además va a **modificar el DataFrame original**.

**Eliminar todas las columnas que tengan todos los registros con valor faltante:**

data\_clean\_empty\_col = data.dropna(**axis = 1**, **how= 'all'**)

**Eliminar todas las filas que tengan todos los registros con valor faltante:**

data\_clean\_empty\_row = data.dropna(**axis = 0**, **how= 'all'**)

**Eliminar todas las columnas con una cantidad de registros con valor (no nulos) menor a un umbral (Threshold):**

Umbral = 13480 (el DataFrame tiene 13580 filas; quiero que como mucho haya 100 registros por columna con valores null; 13580 – 100 = 13480; por eso define como threshold 13480).

data\_clean\_col\_with\_many\_null = data.dropna(**axis = 1**, **thresh=umbral**)

**Eliminar todas las filas con una cantidad de registros con valor (no nulos) menor a un umbral (Threshold):**

umbral = 18 (el DataFrame tiene 21 columnas; quiero que como mucho haya 3 campos por fila con valores null; 21 – 3 = 18; por eso define como threshold 18)

data\_clean\_row\_with\_many\_null = data.dropna(axis = 0, thresh=umbral)

1. **Imputación por Media**: Se reemplazan los valores faltantes por la **media de la serie**, o por la **media condicionada** a determinada categoría. IE: Falta una estatura de una mujer, se la reemplaza por la estatura media de las mujeres.

**Ventajas**: Es muy probable acercarse al valor verdadero del dato faltante. **Desventajas:** Se reduce artificialmente la variabilidad y aleatoriedad de los datos, lo cual puede derivar en conclusiones equivocadas. Se puede ver afectada la correlación entre esta variable en otras, en caso de existir.

data.BuildingArea.fillna(data.BuildingArea.mean(), inplace = True)

1. **Imputación Hot Deck:** Se completa un dato inexistente con algún criterio (aleatorio, secuencial, vecino más cercano).
2. **Imputación por Regresión:** Se emplean modelos de regresión para estimar el dato que falta a partir de las demás variables.

**Introducción a Imputación en Pandas:**

Con el método **fillna()** se pueden hacer varios tipos de imputación que se especifican con el parámetro **method:**

* **method = None** Sirve para completar los datos con valores definidos fuera del dataset. Se ingresa directamente como argumento el valor a imputar.
* **method = pad** o **ffill** Completar los datos faltantes con el valor anterior (sirve mucho para series de tiempo).
* **method = bfill** o **backfill** Completar los datos faltantes con el valor siguiente (sirve mucho para series de tiempo).
* **dff.fillna(dff.mean())** Completar por la media, la moda o la mediana.

**Ejemplo de imputación simple con un valor al azar de la columna**:

Tenemos un DataFrame data con la columna BuildingData con valores de tipo float que tiene Nans.

Se crea un nuevo DataFrame llamado possible values con todos los valores numéricos de la columna BuildingData:

possible\_values = data.BuildingArea[data.BuildingArea.notnull()]

La cantidad de valores posibles es len(possible\_values)

Hace una selección aleatoria de un índice correspondiente a alguno de estos valores posibles y lo asigna en una variable llamada fill\_value\_index

fill\_value\_indexes = rng.integers(low = 0, high = len(possible\_values), size = 1)

fill\_value\_index = fill\_value\_indexes[0]

Por último, genera una variable llamada fill\_value con el valor del DataFrame possible\_values correspondiente al índice recién definido aleatoriamente:

fill\_value = possible\_values.iloc[fill\_value\_index]

Ahora hace la imputación de fill\_value en los nan de BuildingArea generando un nuevo DataFrame usando el método **fillna**:

Data\_BuildingArea\_complete = data.BuildingArea.fillna(fill\_value)

**Ejemplo de imputación simple con un valor determinado**:

data\_car\_2\_fill = data.Car.fillna(2) En este caso rellena los valores NA de la columna Car con un 2.

**Ejemplo de imputación simple con dos valores determinados, imputando un % con un valor y el resto con el otro**:

Se cuentan la cantidad de valores null y la cantidad de los dos valores más probables.

Se genera un nuevo DataFrame con los valores nulos de la columna que queremos generar. Dicho DataFrame mantiene los índices del DataFrame original data:

data\_car\_null\_mask = data.Car.isnull()

data\_car\_null = data.loc[data\_car\_null\_mask, :]

Se genera un segundo nuevo DataFrame con valores aleatorios a partir del DataFrame de datos nulos usando el porcentaje a imputar con uno de los dos valores:

data\_car\_null\_mask\_sample\_1 = data\_car\_null.sample(frac = 0.45).

Se genera un tercer nuevo DataFrame a partir del data\_car\_null\_mask\_sample\_1 con los índices de dichos valores: data\_car\_null\_ones\_index = data\_car\_null\_mask\_sample\_1.index

Para conseguir los índices del resto de los valores, que deben imputarse con el 2do valor se usa el método .index.difference:

data\_car\_null\_twos\_index = data\_car\_null.index.difference(data\_car\_null\_ones\_index)

Por último, usando los índices de los valores null hacemos la imputación directamente sobreescribiendo los valores.

data.loc[data\_car\_null\_ones\_index, "Car"] = 1

data.loc[data\_car\_null\_twos\_index, "Car"] = 2

**Ejemplo de imputación simple con la media de la columna**:

data.BuildingArea.fillna(data.BuildingArea.mean(), inplace = True)

**Ejemplo de Imputación por Media Condicionada**:

Vamos a calcular la media de BuildingArea por CouncilArea e imputar los valores faltantes con esta media condicionada.

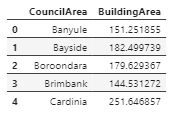
fill\_values = data.groupby(‘CouncilArea’).BuildingArea.mean()

Esto genera una serie con las medias de cada Council como valores y el nombre de cada Council como Index.

Generamos un DataFrame a partir de esta serie para que queden dos columnas (CouncilArea y BuildingArea):

fill\_values\_df = pd.DataFrame(fill\_values)

fill\_values\_df.reset\_index(inplace = True)



Ahora lo combinamos con el DataFrame original para agregar las medias por CouncilArea en una columna llamada BuildingArea\_mean:

data = data.merge(fill\_values\_df, on = “CouncilArea”, suffixes = (“”, “\_mean”), how = ‘left’)

Con fillna completamos los valores de la columna BuildingArea con los valores de la columna BuildingArea\_mean

data.BuildingArea.fillna(data.BuildingArea\_mean, inplace = True)

Al verificar si quedaron valores en nulo con

data.BuildingArea.isnull().sum()

vemos que aún quedan valores en nulo. Esto se debe a que había CouncilAreas con todos los valores de BuildingArea en nulo; entonces no pueden completarse porque no se puede calcular ninguna media para estas áreas.

Si tenemos un grupo de columnas con datos faltantes, podemos analizar individualmente cada una haciendo histogramas gráficos,

o con pd.value\_counts( nombreDataFrame.nombrecolumna).

Si la proporción de valores nulos en la columna (ej 48%); en vez de imputar conviene directamente eliminar la columna. Esto se debe a que aplicar imputaciones en estos casos puede sesgar la información final de forma tal que no tenga nada que ver con la información original, dando lugar a conclusiones erróneas.

**Trabajando con Strings:**

Series e Index cuentan con un conjunto de métodos de procesamiento de cadenas de caracteres que facilitan la operación en cada elemento. Dichos métodos **excluyen automáticamente** los valores **NA**. Se puede acceder a estos métodos con el atributo **.str** y en general coinciden con los métodos sobre strings.

Ejemplo: Tomamos un DataFrame y generamos un nuevo DataFrame a partir del mismo que contenga sólo las columnas de Strings. Para convertirlas efectivamente en strings, usamos astype:

data\_text = data.astype("string")

Con el **método Split** podemos, dado un separador definido, partir un string en varias cadenas de strings:

cadena = 'a,b, guido, asjd, kle, askl'

separador = ','

cadena\_en\_partes = cadena.split(separador)

En este caso por ejemplo cada vez que encuentre una coma, separará el string.

Con el **método strip** podemos conseguir una copia de un string sacando del comienzo y del final de la cadena los caracteres que pasemos como parámetro:

texto = " Este es el primer ejemplo....wow!!! ";

texto\_sin\_espacios = texto.strip()

“Este es el primer ejemplo....wow!!!” (Agrego las comillas para que se vea dónde comienza el texto).

texto1 = "0000000Este es el segundo ejemplo....wow!!!0000000";

texto1\_sin\_ceros = texto1.strip('0')

Este es el segundo ejemplo....wow!!!

texto2 = " 0000000Este es el segundo ejemplo....wow!!!0000000";

texto2\_sin\_ceros = texto2.strip('0')

0000000Este es el segundo ejemplo....wow!!!

Con el **método find**, devuelve el mínimo índice donde se encuentra el substring que le demos como argumento. En caso de no encontrar, devuelve -1.

IE: nombrestring.find(':')

El **método index**, simil find, pero devuelve una excepción de tipo **ValueError** en caso de no encontrar el substring.

El **método Count** devuelve cuántas veces aparece un substring en un string:

IE: nombrestring.count(‘,’) Me va a indicar cuántas comas hay en el string nombrestring.

El **método replace** da una copia del string original que reemplace lo que informemos como primer argumento con lo que tenga el segundo argumento.

IE: cadena\_punto\_y\_coma = cadena.replace(',', ';') va a buscar todas las comas y las va a reemplazar por punto y coma.

**Trabajando con texto en Pandas:**

NombreDataFrame.Nombrecolumna**.str.lower()** o bien nombreserie**.str.lower()** baja todo el string a minúsculas.

NombreDataFrame.Nombrecolumna**.str.upper()** o bien nombreserie**.str.uppper()** baja todo el string a mayúsculas.

NombreDataFrame.Nombrecolumna**.str.len()** o bien nombreserie**.str.len()** devuelve la cantidad de caracteres del string.

Estas 3 operaciones respetan el tipo de datos original del DataFrame o Series: Si es un object, devuelven object; si es un string, devuelve un string.

NombreDataFrame.Nombrecolumna**.str.strip()** Nombreseries**.str.strip()** borra los espacios que rodean a una cadena de caracteres.

NombreDataFrame.Nombrecolumna**.str.lstrip()** Nombreseries**.str.lstrip()** borra los espacios a la izquierda de una cadena de caracteres.

NombreDataFrame.Nombrecolumna**.str.rstrip()** Nombreseries**.str.rstrip()** borra los espacios a la derecha de una cadena de caracteres.

NombreDataFrame.Nombrecolumna**.str.split(caracterseparador)** Nombreseries**.str.split(caracterseparador)** Devuelve un array donde cada elemento es una subcadena del original; separada por el carácter especificado. Agregando al final .get[#] se puede llamar a un elemento específico del array, usando la posición numérica del índice.

NombreDataFrame.Nombrecolumna**.str.replace(stringareemplazar, stringquereemplaza, case= False, regex = False)** Nombreseries**.str. replace(stringareemplazar, stringquereemplaza, case= False, regex = False)** reemplaza el string del primer argumento por el string del segundo argumento. Es posible usar expresiones regulares para definir el patrón a reemplazar.

NombreDataFrame.Nombrecolumna**.str.extract(regladeexpresiónregular, expand = True/False)** Nombreseries**.str.extract()** devuelve un DataFrame o una Serie con las subcadenas que cumplan con el patrón especificado. Si **expand** **= True** devuelve un DataFrame; si **expand** **= False** devuelve una Serie.

**Tidy Data:**

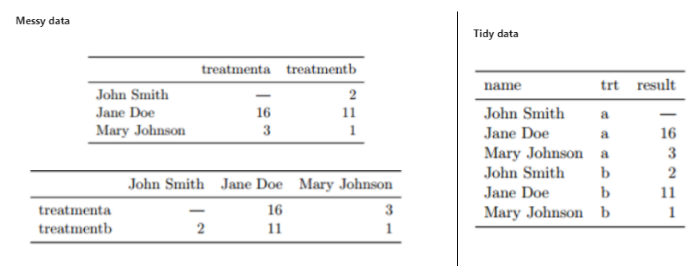
Reglas para que un dataset esté ordenado**:**

* Cada variable es una columna
* Cada observación es una fila
* Cada tipo de unidad observacional forma una tabla

**Variable:** Medición de un atributo (peso, altura)

**Valor:** Medida que toma una variable para una observación

**Observación:** Todas las observaciones toman el mismo tipo de valores para cada variable.



En Tidy data, cada fila tiene un tratamiento que se dio una persona determinada, sin valores null, como ocurre en los ejemplos messy data donde se cruzan personas con tratamientos que nunca se dieron.

**Herramientas para la Limpieza de Datos:**

**Expresiones Lambda**: Las funciones lambda son funciones que se definen en el código que las va a usar sin necesidad de darles un nombre (funciones anónimas). Se definen en una única línea, con una única expresión y no contienen return.

IE: square = lambda x: x \*\* 2

Square(10)

**Método apply():** Con este método se puede aplicar cualquier función a los elementos de un DataFrame. Es posible utilizarlo por columna, por fila o elemento a elemento:

1. **Por Columna:** df.apply(mifunción) Va a hacer una iteración por filas.
2. **Por Fila:** df.apply(mifunción, axis = 1) Va a hacer una iteración por columnas.
3. **Elemento a Elemento:** df.applymap(mifunción).

También puede aplicarse sobre Series, donde iterará elemento a elemento, ya sea con el método .apply o con el método .map; el resultado es similar con cualquiera de estos 2 métodos; la diferencia radica en que con .map se puede generar un DataFrame a partir de la Serie; mientras que con .apply se puede crear una serie de series.

Permite aplicar las **operaciones vectorizadas** de **Numpy**.

IE: Dado un DataFrame data, usando el método .apply y una función lambda, calcula el valor por metro cuadrado para aquellos casos en los que el metro cuadrado no es 0.

Precio\_m2 = data.apply(lambda x: 0 if x[‘Landsize’] == 0 else x[‘Price’] / x[‘Landsize’], axis = 1)

**Expresiones Regulares:**

Una **expresión regular** (Regex) es una **secuencia de caracteres** que definen un **patrón de búsqueda** de texto. Tienen un lenguaje flexible que sirve para identificar y extraer información de un **cuerpo de caracteres no estructurado.**

IE: Identificar mails, o DNI, CUIL, CUIT, etc.

Con el módulo re de Python se pueden aplicar expresiones regulares a cadenas de caracteres. Las funciones de re se pueden agrupar en 3 categorías:

1. **Split**: Divide una cadena de caracteres, usando el patrón definido por la expresión regular como separador.
2. **Pattern Matching**: Extrae de una cadena de caracteres las subcadenas definidas por el patrón de expresión regular.
3. **Sustitución**: reemplaza la cadena de caracteres definida por la expresión regular por otra cadena de caracteres.

**Metacaracteres especiales:** Aquellos que indican que se debe matchear algo de forma **no literal** o que afectan a otras partes de la regex repitiendo caracteres o cambiando su significado. Ejemplos de algunos:

**\d** cualquier dígito del 0 al 9

**\w** cualquier carácter alfanumérico (A-Z, a-z, 0-9 y \_)

**\s** cualquier espacio en blanco (espacio, tabulado, nueva línea, entre otros)

**.** cualquier carácter excepto nueva línea (**\n**)

**\D** Todo menos cualquier dígito.

**\W** Todo menos cualquier carácter alfanumérico.

**\S** Todo menos cualquier espacio en blanco.

**[ ]** Definen un conjunto de caracteres.

**[A-Za-z0-9\_]** Definen rangos.

**()** Definen un grupo de captura.

**|** Operador “o”.

**\** Escapa los caracteres especiales.

**Metacaracteres Cuantificadores:**

**\*** Cero o más del elemento anterior.

**+** Uno o más del elemento anterior.

**{m}** indica exactamente m repeticiones.

{**min, max**}Definen mínimos y máximos de repetición.

**?** Opcional, o cero o uno del elemento anterior.

**Metacaracteres de Posición:**

^ Comienzo de un String (o negación de comienzo de un conjunto, [^ ])

**$** Final de un String

**\b** Límite de palabra.

**Opciones:**

**[abc]** Indica un carácter perteneciente al conjunto de valores posibles especificados entre corchetes.

**[a-z]** Indica un carácter perteneciente al intervalo de valores especificados entre corchetes.

**Grupos**:

**()** Define un grupo

**(?P<group\_name>)** Define un grupo etiquetado.

IE:

**\d+** Encuentra un número o más.

**.\*** Encuentra cadenas de caracteres de cualquier longitud, incluyendo vacías.

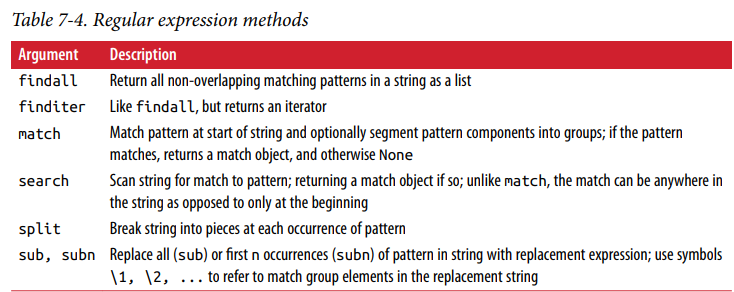
**\w{2,6}** Encuentra un conjunto de caracteres alfanuméricos con una longitud que va de 2 a 6 caracteres.

**[a-zA-Z]** o bien **[a-zA-Z]{1}** encuentra cualquier carácter entre a y z y/o entre A y Z.

**[a-zA-Z]+** encuentra cadenas de caracteres entre a y z y/o entre A y Z de longitud de al menos 1.

**(\d\d\d\d)** o bien **\d4** encuentra cadenas de 4 dígitos.

**(?P<num>\d\d\d\d)** encuentra cadenas de 4 dígitos, a las cuales etiqueta como Num.

****

Si queremos identificar direcciones de mails:

import re

patron = “^(?P<nombre\_de\_usuario>.\*)@(?P<dominio>.\*)(\.com)(\.ar)?$”

regex = re.compile(patron, flags = re.IGNORECASE)

texto1 = “ana\_laura@hotmail.com”

regex.search(texto1).groups()

devuelve: (ana\_laura, Hotmail, .com, None)

regex.search(texto1).group(0)

devuelve: ‘ana\_laura@hotmail.com’

regex.search(texto1).group(1) o regex.search(texto1).group(“nombre\_de\_usuario”)

devuelve: ‘ana\_laura’

regex.search(texto1).group(2) o regex.search(texto1).group(“dominio”)

devuelve: ‘hotmail’

regex.search(texto1).group(3)

devuelve: ‘.com’

regex.search(texto1).group(4)

No devuelve nada.

**Ejemplo de Split:**

patron = "\s+"

regex = re. Compile(patron, flags = re.IGNORECASE)

Lo que va a hacer es separar el texto cada vez que encuentre un conjunto de espacios entre strings.

**Ejemplos de Pattern Matching:**

Si en vez de quedarnos con el texto nos queremos quedar con **todas las subcadenas** que verifiquen el patron de la expresión regular (en el ejemplo "\s+" es decir, los espacios):

**regex.findall**(texto)

En caso de querer **sólo el primer match** de la expresión regular:

**regex.search**(texto)

En caso de querer verificar si el patrón de la expresión regular se encuentra **al comienzo del texto**:

**regex.match**(texto)

**Ejemplos de Sustitución:**

Si queremos reemplazar todas las subcadenas que verifican el patrón de la expresión regular por “ [espacios] “

**regex.sub**(‘ [espacios] ‘, texto)

Entonces cada vez que encuentre una subcadena de espacios la va a reemplazar por ‘ [espacios] ‘.

**Ejemplo Integrador de Regex:**

Dado un dataframe con una columna que tiene direcciones, se pide extraer los números, las calles y sacar la terminación St.

Se genera un DataFrame que contenga sólo la columna con las direcciones:

address\_series = data.Address

Se define un patrón que detecte números:

patron\_num = "\d+"

patron\_num\_regex = re.compile(patron\_num)

Usando .apply() y una función lambda con el comando findall, aplicamos la regex a todas las filas del DataFrame recién creado:

resultado = address\_series.apply(lambda x: patron\_num\_regex.findall(x))

Como resultado conseguimos una serie con una lista de los números de cada dirección, en cada elemento.

Si en vez de usar findall usamos search en la expresión lambda, obtenemos una serie con el primer número que encontró en cada dirección, guardado como un **objeto match**.

Podemos extraer el valor del número Match usando **group** o **[]**:

numeros\_match = resultado.apply(lambda x: x[0])

o bien,

numeros\_match = resultado.apply(lambda x: x.group(0))

Ahora es como que vuelve a empezar:

Genera un patrón:

Identifica el número de la calle con ?P<numero\_calle>\d+[a-z]\*) Con esto lo que está diciendo es que busque uno o más dígitos, posiblemente seguidos por caracteres de la a a la z.

Luego aplica un separador de espacios \s

A continuación, identifica el nombre de la calle buscando cualquier carácter, pero exigiendo que haya por lo menos uno (?P<nombre\_calle>.+)

Por último, busca el sufijo al final encontrando un separador de espacios y exigiendo que lo que le sigue sea alguno de los 4 sufijos posibles (?P<sufijo>\sSt|La|Cr|Dr)

pattern\_nombre = "(?P<numero\_calle>\d+[a-z]\*)\s(?P<nombre\_calle>.+)(?P<sufijo>\sSt|La|Cr|Dr)"

pattern\_nombre\_regex = re.compile(pattern\_nombre)

Ahora usa apply, combinado con una función lambda y el comando search, usando el nuevo patrón generado:

resultado\_nombres = address\_series.apply(lambda x: pattern\_nombre\_regex.search(x))

Se puede extraer el nombre de la calle del objeto Match que sale de la sentencia anterior, usando **group** y sabiendo que el nombre del grupo es **nombre\_calle**. Nuevamente acudimos al uso de apply combinado con una función lambda para hacer esto sobre todos los registros del objeto match:

calles\_match = resultado\_nombres.apply(lambda x: x if x is None else x.group('nombre\_calle'))

Otro approach posible es, usando el mismo patrón pattern\_nombre, separar la columna address en tres partes: número, nombre y sufijo. Esto lo logramos recurriendo una vez más al método apply con una función lambda que aplique la operación Split sobre todos los registros de la serie address:

resultado\_nombres = address\_series.apply(lambda x: pattern\_address\_regex.split(x))

Por último, como no queremos los sufijos, podemos reemplazarlos por “”. Para esto creamos un patrón que los encuentre:

pattern\_suffix = "\sSt|La|Cr|Dr"

pattern\_suffix\_regex = re.compile(pattern\_suffix)

Y una vez más usamos el método apply sobre la serie address, usando una función lambda que use regex.sub para reemplazarlos por “”:

resultado\_reemplazo = address\_series.apply(lambda x: pattern\_suffix\_regex.sub(“”, x))