Trabajando con Datos. Procesado

Procesado

En esta unidad trabajaremos con procesados adicionales para datos que a veces son necesarios:

- cadenas de texto
- integración de fuentes externas de datos
- transformación de columnas para dar mayor valor a los datos o creación de nuevas

Un ejemplo sería un fichero de compras en el cual aparece la fecha de nacimiento se podría transformar en su edad, ya que facilita la manipulación en herramientas BI.

Otro caso sería en los sueldos de los trabajadores aspectos como tickets de comida o de transporte, transformarlos en cantidades numéricas de su valor.

Aunque las herramientas BI pueden hacer transformaciones resulta más eficiente realizarlas en el momento de la carga de datos.

No existen criterios generales de aplicación para transformar datos dependen siempre del tipo de proyecto y los requisitos que se planteen desde otros departamentos.

Procesado

Veremos métodos básicos en Python sobre dataframes *(df_compas y df_clientes)*

El resultado se almacenará en una bbdd que consultarán los usuarios. Usaremos: Activity. A.3.2.procesado.ipynb

Haremos primero un repaso general de los métodos más utilizados.

A. Métodos útiles para manipulación de cadenas: *cadena.método(parámetros)*

split	Divide una cadena en una lista de subcadenas separadas por el separador indicado (por defecto el espacio).
strip	Elimina espacios al principio y final de la cadena
join	Une los elementos indicados en una sola cadena, separados por el separador indicado (hace lo opuesto a split)
index	Devuelve la primera posición en la que se encuentra el substring
find	Devuelve la posición de la primera ocurrencia del substring
rfind	Devuelve la posición de la última ocurrencia del substring
count	Devuelve el número de veces que aparece el substring
replace	Reemplaza un substring por otro

Procesado

Ejemplos.

.split()

mi_cadena = "Yo programo dos horas al día" *print(mi_cadena.split())* ⇒ `['Yo', 'programo', 'dos', 'horas', 'al', 'día']`

mi_cadena = "Yo programo: dos horas al día" *print(mi_cadena.split(":"))* ⇒ `['Yo programo', ' dos horas al día']`

.strip()

string = ' Hola, Mundo! ' *print(string.strip())* ⇒

```
Hola, Mundo!
```

.join()

numeros = ['1', '2', '3', '4', '5'] *print('-'.join(numeros))* ⇒

```
1-2-3-4-5
```

Procesado

.index y .find()

cadena='Esto es una cadena de caracteres' *cadena.index('es')* ⇒ `5` *cadena.find('es')* ⇒ `5`

Aunque parece que se comportan igual, no es así en el caso de no encontrar coincidencia, por ejemplo:

cadena.index('para') ⇒

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

(excepción) pero *cadena.find('para')* ⇒ `-1`

.rfind() *(igual que find pero devuelve la última posición, sino existe devuelve -1)*

cadena.rfind('una') ⇒ `8`

.count()

cadena.count('es') ⇒ `2`

.replace()

cadena.replace('es', 'era', 1) ⇒ `'Esto era una cadena de caracteres'` ¿por qué ponemos 1?

Procesado

B. Expresiones regulares

El uso de expresiones regulares permiten ir un paso más allá en el procesado de cadenas de texto por su versatilidad.

Necesitamos importar la librería ***re***

split	Divide la cadena en base a la expresión regular
findall	Devuelve las ocurrencias que cumplen el patrón en forma de lista
finditer	Como el anterior, pero devuelve un iterable
match	Busca correspondencia de principio a fin
search	Devuelve la primera subcadena que cumpla el patrón
sub	Reemplaza las ocurrencias de patrones por una expresión
compile	Permite compilar la expresión regular. Lo veremos con ejemplos

Procesado

.split() separa una cadena en subcadenas utilizando como separador una expresión regular. Imagina string separados por #

```
cadena = 'Cadena#que##separa###las####palabras#####de#forma##no###uniforme'

#Si no usamos expresiones regulares
print(cadena.split('#'))
```

```
['Cadena', 'que', '', 'separa', '', '', 'las', '', '', '', 'palabras', '', '', '', '', 'de', 'forma', '', 'no', '', '', 'uniforme']
```

```
import re
re.split('#+', cadena)
```

```
['Cadena', 'que', 'separa', 'las', 'palabras', 'de', 'forma', 'no', 'uniforme']
```

- '#+' significa 1 o más '#'
- Prueba a ejecutarlo con '#*'

.compile() puedes definir una expresión regular que después se reutiliza

re.split('#+', 'Primera#cadena') ⇒ `['Primera', 'cadena']` *re.split('#+', 'Segunda#cadena')* ⇒ `['Segunda', 'cadena']`

Pero si creamos el patrón (r = raw para caracteres raros, no es este caso)

expresion.split('Primera#cadena') ⇒ `['Primera', 'cadena']` expresion.split('Segunda#cadena') ⇒ `['Segunda', 'cadena']`

Procesado

.findall() Imaginemos la siguiente lista *agenda_mails='''*

Juan:juan#juan.es

María:maria@maria.es

Pedro:pedro@pedro.es

'''

El siguiente patrón es el que corresponde al formato de correos electrónicos:

patron = r'[A-Z0-9._%+-]+@[A-Z0-9._]+\.[A-Z]{2,4}' *(letras y/o números y determinados caracteres,luego @, luego letras y números, luego punto, y luego letras entre 2 y 4, \. es necesario sino interpreta no un punto, sino cualquier otra coas)*

Compilamos ignorando mayúsculas ⇒ *expresion = re.compile(patron, flags=re.IGNORECASE)*

La siguiente expresión nos busca solo valores que cumplan la condición del patrón.

expresion.findall(agenda_mails) ⇒ `['maria@maria.es', 'pedro@pedro.es']`

Si buscamos segmentar (fíjate los paréntesis) *patron = r'([A-Z0-9._%+-]+)@([A-Z0-9._]+)\.([A-Z]{2,4})'*

Compilamos ignorando mayúsculas ⇒ *expresion = re.compile(patron, flags=re.IGNORECASE)*

xpresion.findall(agenda_mails) ⇒ `[('maria', 'maria', 'es'), ('pedro', 'pedro', 'es')]`

Procesado

.search() *d*evuelve un objeto de tipo Match con la primera ocurrencia del patrón.

expresion.search(agenda_mails) ⇒ `<re.Match object; span=(44, 58), match='maria@maria.es'>`

.match() busca el patrón, pero debe coincidir de principio a fin

print(expresion.match(agenda_mails)) ⇒ `None`

.sub() sustituye el patrón por la cadena indicada.

expresion.sub("ELIMINADO", agenda_mails) ⇒ `Juan:juan#juan.es\n María:ELIMINADO\n Pedro:ELIMINADO\`

expresion.sub("ELIMINADO", agenda_mails, count=1) *(si solo queremos 1)* ⇒ `Juan:juan#juan.es\n María:ELIMINADO\n Pedro:pedro@pedro.`

Si previamente dividimos en componentes, podemos acceder a cada uno mediante \1, \2, etc.

print(expresion.sub(r'Usuario: \1, Dominio: \2, Dom_raiz: \3', agenda_mails)) ⇒

```
Juan:juan#juan.es
María:Usuario: maria, Dominio: maria, Dom_raiz: es
Pedro:Usuario: pedro, Dominio: pedro, Dom_raiz: es
```

Procesado – Pandas

.map() nos permite tratar cadenas en dataframe

```
df_correos = pd.DataFrame({
    'Nombre': ['Juan', 'María', 'Pedro'],
    'Apellidos': ['Fernández', 'Martínez', 'Álvarez'],
    'email': ['juan#juan.es', 'maria@maria.es', 'pedro@pedro.es']
})
```

	Nombre	Apellidos	email
0	Juan	Fernández	juan#juan.es
1	María	Martínez	maria@maria.es
2	Pedro	Álvarez	pedro@pedro.es

Si queremos sustituir el dominio .es por .com

```
df_correos['email'] = df_correos['email'].map(lambda cadena: cadena.replace('.es','.com'))
```

	Nombre	Apellidos	email
0	Juan	Fernández	juan#juan.com
1	María	Martínez	maria@maria.com
2	Pedro	Álvarez	pedro@pedro.com

O los que hicimos con valor DESCONOCIDO

```
#Para facilitar legibilidad definimos el patrón
patron = r'[A-Z0-9._%+-]+@[A-Z0-9._]+\.[A-Z]{2,4}'

#Reseteamos aquellos valores que siguen patrón de email con la cadena "DESCONOCIDO"
df_correos['email'] = df_correos['email'].map(
    lambda cadena: re.sub(patron, "DESCONOCIDO", cadena, flags=re.IGNORECASE)
)
```

	Nombre	Apellidos	email
0	Juan	Fernández	juan#juan.es
1	María	Martínez	DESCONOCIDO
2	Pedro	Álvarez	DESCONOCIDO

Procesado – Pandas

Pero tiene un problema falla con valores NaN, para ello hay que recorrerlo y evitando dichos valores, utilizando el atributo str, así:

```
df_correos = pd.DataFrame({
    'Nombre': ['Juan', 'María', 'Pedro'],
    'Apellidos': ['Fernández', 'Martínez', 'Álvarez'],
    'email': [np.NaN, 'maria@maria.es', 'pedro@pedro.es']
})
```

	Nombre	Apellidos	email
0	Juan	Fernández	NaN
1	María	Martínez	maria@maria.es
2	Pedro	Álvarez	pedro@pedro.es

```
df_correos['email'] = df_correos['email'].str.replace('.es','.com')
```

	Nombre	Apellidos	email
0	Juan	Fernández	NaN
1	María	Martínez	maria@maria.com
2	Pedro	Álvarez	pedro@pedro.com

Procesado – Índice jerárquicos y multi-índices

Estructurar el dataframe en varios niveles de índices puede ser muy útil, ya que permite, por ejemplo, hacer consultas por uno de los niveles.

Por ejemplo nos puede interesar los ventas por ID Cliente o las ventas por el ID pedido para ver tendencias de la compras realizadas por el cliente.

		Importe	Pagado
IDCliente	**IDPedido**		
ID1	**PID1**	3.56	SI
	PID2	7.45	NO
ID2	**PID1**	23.89	SI
	PID2	101.34	SI
	PID3	204.01	NO
ID3	**PID1**	45.34	NO

Procesado – Índice jerárquicos y multiíndices

```
df_ejemplo.loc['ID2']
```

IDPedido	Importe	Pagado
PID1	23.89	SI
PID2	101.34	SI
PID3	204.01	NO

Filtra el índice de nivel 0

```
df_ejemplo.loc['ID2','PID3']
```

```
Importe    204.01
Pagado         NO
Name: (ID2, PID3), dtype: object
```

En este caso filtramos por ambos índices.

```
df_ejemplo.loc['ID2':'ID3']
```

IDCliente	IDPedido	Importe	Pagado
ID2	PID1	23.89	SI
	PID2	101.34	SI
	PID3	204.01	NO
ID3	PID1	45.34	NO

Pedidos de clientes empezando en el ID2 y terminando en el ID3

Procesado – Índice jerárquicos y multi-índices

```
df_ejemplo.loc['ID2','Importe']
```

```
IDPedido
PID1     23.89
PID2    101.34
PID3    204.01
Name: Importe, dtype: float64
```

Filtramos por el índice de nivel 0 y la columna 'Importe'.

```
df_ejemplo['Importe'].loc[:,'PID2']
```

```
IDCliente
ID1      7.45
ID2    101.34
Name: Importe, dtype: float64
```

Nos quedamos con la serie 'Importe' y devolvemos el importe de los pedidos PID2 de cualquier cliente.

Procesado – Combinación de dataframes

A veces se hace necesario combinar dataframes

Pandas dispone de varios métodos para combinar información:

- ***merge***: combina dos dataframes en base a una clave. Es equivalente a los joins en bbdd relacionales.

- ***concat:*** concatena datos en el eje de filas o de columnas.

- ***combine_first:*** rellena valores NaN de un dataframe con valores que estén en la misma posición del segundo dataframe

Procesado – Combinación de dataframes

Los ***joins*** deben hacerse en base a un columna o índice común.

- columna. Si ejecutamos *pd.merge(df_edades,df_alturas)*

	nombre	edad
0	Juan	10
1	María	12
2	Pedro	20
3	Patricia	23
4	Ana	5

	nombre	edad	altura
0	Juan	10	100
1	María	12	120

⇒

	nombre	edad	altura
0	Juan	10	100
1	María	12	120

- Solo obtenemos datos para los nombres que existen en ambos dataframes. Por defecto, *.merge hace un inner join.*
- No hemos necesitado indicarle la columna que tiene la clave, ya que ambas tienen una columna que se llama igual.

Procesado – Combinación de dataframes

Si necesitamos indicarle el nombre de la columna: ***pd.merge(df_edades,df_alturas, on='nombre')***

	nombre	edad	altura
0	Juan	10	100
1	María	12	120

Si las columnas no se llaman igual entonces usamos los parámetros ***left_on*** (columna 1ºtabla) y ***right_on*** (columna 2ºtabla)

El problema es que duplica la columna lo que en un proceso posterior se elimina.

El *join* se puede hacer sobre múltiples columnas. En ese caso, debes especificar las columnas en formato lista

(on = [columna1, …, columnaN])

Procesado – Combinación de dataframes

Si ambas tablas tienen columnas que se llaman igual pero sobre las que NO hacemos join

```
#Añadimos una columna a cada dataframe que se llame igual
df_edades['País'] = ['España', 'Francia', 'Alemania', 'Francia', 'España']
df_alturas['País'] = ['España', 'Francia']

pd.merge(df_edades, df_alturas, left_on='nomEdad', right_on='nomAltura')
```

	nomEdad	edad	País_x	nomAltura	altura	País_y
0	Juan	10	España	Juan	100	España
1	María	12	Francia	María	120	Francia

Pandas añade un sufijo (_x, _y). Puedes indicar el sufijo mediante la opción ***suffixes***

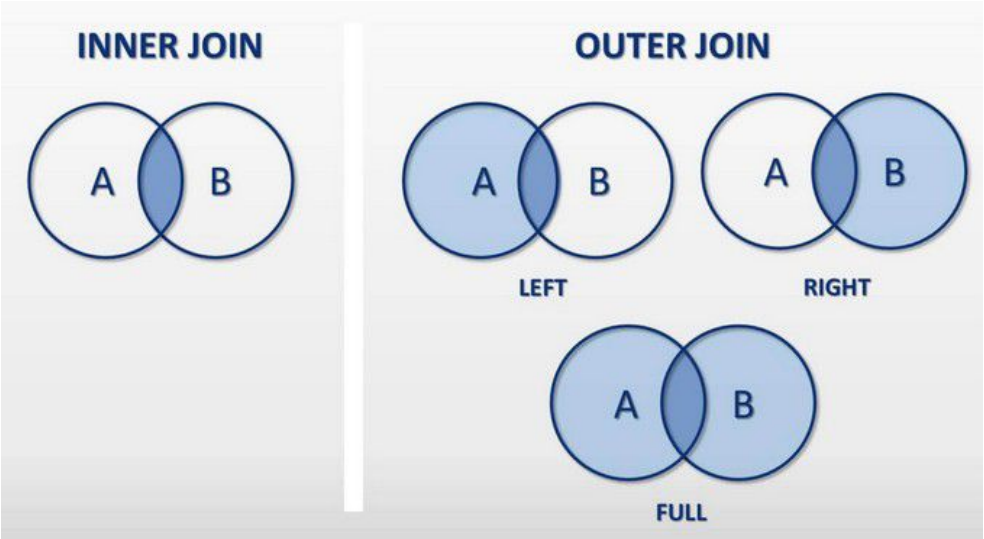
pd.merge(df_edades, df_alturas, left_on='nomEdad', right_on='nomAltura', suffixes=('_primera','_segunda'))

	nomEdad	edad	País_primera	nomAltura	altura	País_segunda
0	Juan	10	España	Juan	100	España
1	María	12	Francia	María	120	Francia

Procesado – Combinación de dataframes

Pandas permite también hacer *left outer join, right outer join y full outer join.* Para ello, se añade la opción ***how,*** con los valores:

- inner: ***inner join.*** Es la opción por defecto si no se especifica nada.
- left: ***left outer join***
- right: ***right outer join***
- output: ***full outer join***



Procesado – Combinación de dataframes

pd.merge(df_edades, df_alturas, left_on='nomEdad', right_on='nomAltura',

suffixes=('_primera','_segunda'), how='left')

	nomEdad	edad	País_primera	nomAltura	altura	País_segunda
0	Juan	10	España	Juan	100.0	España
1	María	12	Francia	María	120.0	Francia
2	Pedro	20	Alemania	NaN	NaN	NaN
3	Patricia	23	Francia	NaN	NaN	NaN
4	Ana	5	España	NaN	NaN	NaN

pd.merge(df_edades, df_alturas, left_on='nomEdad', right_on='nomAltura',

suffixes=('_primera','_segunda'), how='right')

"coincide con la anterior porque los datos de la columna segunda también están en la primera"

	nomEdad	edad	País_primera	nomAltura	altura	País_segunda
0	Juan	10	España	Juan	100	España
1	María	12	Francia	María	120	Francia

IMPORTANTE: Al hacer el merge en base a columnas, los índices de las filas se pierden. Esto no sucede con merge en base a índices.

Procesado – Combinación de dataframes

El modelo soporta relaciones varios a varios, en cuyo caso el número de filas en el dataframe resultado aumenta (*será un producto cartesiano).*

```
df_edades = pd.DataFrame({
    'nombre':['Juan','María','Juan','Patricia','Ana'],
    'edad': [10, 12, 20, 23, 5]
})
df_alturas = pd.DataFrame({
    'nombre':['Juan','María','Juan'],
    'altura': [100, 120, 150]
})
pd.merge(df_edades,df_alturas)
```

```
pd.merge(df_edades,df_alturas)
```

	nombre	edad	altura
0	Juan	10	100
1	Juan	10	150
2	Juan	20	100
3	Juan	20	150
4	María	12	120

Procesado – Combinación de dataframes

Supongamos ahora que la clave del join es el índice de las filas

```
df_edades = pd.DataFrame(
    [[10, 'España'], [12, 'Francia'], [20,'Alemania' ], [23, 'Francia'], [5, 'España']],
    index = ['Juan','María','Pedro','Patricia','Ana'],
    columns = ['edad', 'país']
)
df_alturas = pd.DataFrame(
    [[100, 'España'],[120, 'Francia']],
    index = ['Juan','María'],
    columns = ['altura','país']
)
```

	edad	país
Juan	10	España
María	12	Francia
Pedro	20	Alemania
Patricia	23	Francia
Ana	5	España

	altura	país
Juan	100	España
María	120	Francia

pd.merge(df_edades, df_alturas, left_index=True, right_index=True)

	edad	país_x	altura	país_y
Juan	10	España	100	España
María	12	Francia	120	Francia

Debemos utilizar *right_index = True* y/o *left_index=True*

Procesado – Combinación de dataframes

Pandas dispone de otro método para realizar joins por el índice: ***.join***

```
df_edades = pd.DataFrame(
    [[10, 'España'], [12, 'Francia'], [20,'Alemania' ], [23, 'Francia'], [5, 'España']],
    index = ['Juan','María','Pedro','Patricia','Ana'],
    columns = ['edad', 'país']
)
df_alturas = pd.DataFrame(
    [[100, 'España'],[120, 'Francia']],
    index = ['Juan','María'],
    columns = ['altura','país']
)
```

	edad	país
Juan	10	España
María	12	Francia
Pedro	20	Alemania
Patricia	23	Francia
Ana	5	España

	altura	país
Juan	100	España
María	120	Francia

df_edades.join(df_alturas, lsuffix='_primera')

Cambia el modo de invocar ejecutando el método sobre la instancia del dataframe. Por defecto el izquierdo, pero se puede cambiar con ***how*** que no veremos

⇒

	edad	país_primera	altura	país
Juan	10	España	100.0	España
María	12	Francia	120.0	Francia
Pedro	20	Alemania	NaN	NaN
Patricia	23	Francia	NaN	NaN
Ana	5	España	NaN	NaN

VUESTRO TURNO