

MANIPULACIÓN DE DATOS

UNIDAD 2

Fundamentos de Ciencia de Datos 2025



DATA WRANGLING

Es el proceso de preparar los datos y ponerlos en el formato necesario para poder realizar un posterior análisis de los mismos.

Data sets
in tutorials



Data sets in
the wild



FORMA LARGA/FORMA ANCHA

Reformar un **DataFrame** de **pandas** es una de las tareas de manipulación de datos más comunes en el mundo del análisis de datos y consiste en su transposición desde un **formato ancho** (*wide*) a uno **largo** (*long*) o viceversa.

DATOS EN FORMA LARGA/FORMA ANCHA

FORMA ANCHA

En este caso, la columna que identifica al dato no tiene valores repetidos (**i**d).

id	tiempo_auto	tiempo_moto	tiempo_bus	tiempo_bici	modo_elegido
1	10	8	15	20	bici
2	20	15	45	50	auto

DATOS EN FORMA LARGA/FORMA ANCHA

FORMA LARGA

La columna que identifica al registro tiene valores repetidos y ya no puede utilizarse por sí misma como identificación inequívoca del registro, sino que debe combinarse con otra columna, en este caso, **modo**.

id	modo	tiempo	modo_elegido
1	auto	10	bici
1	moto	8	bici
1	bus	15	bici
1	bici	20	bici
2	auto	20	auto
2	moto	15	auto
2	bus	45	auto
2	bici	50	auto

DATOS EN FORMA LARGA/FORMA ANCHA

Para pasar nuestra tabla de formato ancho a formato largo podemos utilizar la operación `pd.melt()` de `pandas`, la cual nos permite agrupar varias columnas en una sola, produciendo un **DataFrame** que es más largo que el de partida.

DATOS EN FORMA LARGA/FORMA ANCHA

Simulamos algunos datos para el ejemplo:

```
import pandas as pd
import random

# Generamos datos "de juguete" para el ejemplo que están en formato ancho
data = pd.DataFrame(index = range(100), columns=['tiempo_auto', 'tiempo_moto', 'tiempo_bus', 'tiempo_bici', 'modo_elegido'])
data['tiempo_auto'] = [random.randint(1,100) for x in range(0,100)]
data['tiempo_moto'] = [random.randint(1,100) for x in range(0,100)]
data['tiempo_bus'] = [random.randint(1,100) for x in range(0,100)]
data['tiempo_bici'] = [random.randint(1,100) for x in range(0,100)]
data['modo_elegido'] = [random.randint(1,4) for x in range(0,100)]
data.index.name = 'person_id'
```

	person_id	tiempo_auto	tiempo_moto	tiempo_bus	tiempo_bici	modo_e
0	0	1	78	87	51	1
1	1	13	21	67	57	4

DATOS EN FORMA LARGA/FORMA ANCHA

Convertimos a formato largo usando `pd.melt()` y limpiamos la columna `modo` para que sólo contenga el medio de transporte.

```
import pandas as pd
import random

# Generamos la tabla en formato largo usando el método melt()
df = pd.melt(data, id_vars = ['person_id', 'modo_elegido'], value_vars =

# Corregimos la información presentada en la columna 'modo'
df['modo'] = df['modo'].str.replace('tiempo_', '')
```


DATOS EN FORMA LARGA/FORMA ANCHA

```
# Mostramos dataset en formato largo
print(df)
```

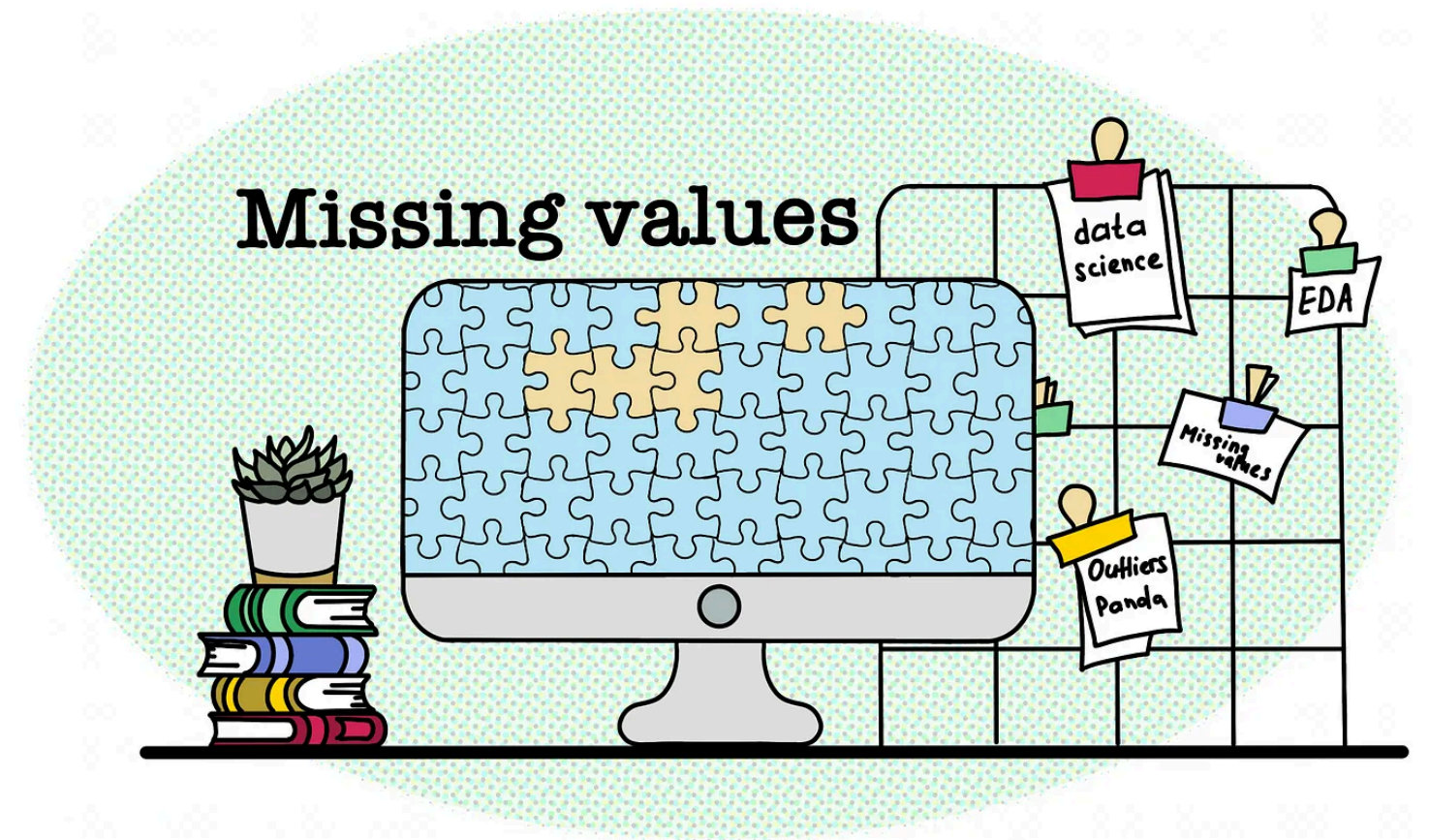
	modo_elegido	modo	tiempo
person_id			
0	1	auto	1
1	4	auto	13
2	2	auto	18
3	2	auto	24
4	1	auto	8
...
95	3	bici	6
96	1	auto	70

¿Por qué el DataFrame resultante tiene 400 filas si sólo contamos con la info de 100 personas?

MANEJO DE DATOS FALTANTES

Es común que en el análisis de datos nos encontremos con columnas que presentan **datos faltantes** (los famosos NaN).

Su presencia en los datasets puede venir ligada a errores en la recolección de datos o en la elaboración de la base, o simplemente están allí porque no es esperable que el registro tenga un valor para esa columna.



MANEJO DE DATOS FALTANTES

¿Qué podemos hacer con ellos?

A grandes rasgos, podemos hacer dos cosas con los datos faltantes:

- Eliminar los registros que los contienen
- **Imputarlos**, es decir, reemplazarlos por algún valor elegido **de acuerdo a ciertos criterios**

ELIMINAR REGISTROS CON DATOS FALTANTES

Por *default*, el método **dropna()** elimina cualquier fila del **DataFrame** que contenga, al menos, un valor faltante.

```
import numpy as np
import pandas as pd

# Definimos DataFrame "de juguete"
data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan], [np.nan, np.nan, np.nan]],
                    columns = ['ColA', 'ColB', 'ColC'])

# Vemos cómo luce nuestro set de datos
print(data)
```

	ColA	ColB	ColC
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

ELIMINAR REGISTROS CON DATOS FALTANTES

¿Cuál es el resultado de aplicar **dropna()** sobre el DataFrame?

```
data_dropped = data.dropna()
```

```
print(data_dropped)
```

	ColA	ColB	ColC
0	1.0	6.5	3.0

ELIMINAR REGISTROS CON DATOS FALTANTES

Si agregamos el argumento **how = 'all'**, eliminaremos únicamente aquellos registros que están formados por completo por **NaN**.

```
data_dropped = data.dropna(how = 'all')
```

```
print(data_dropped)
```

	ColA	ColB	ColC
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

Si estamos interesados en realizar la misma operación **por columnas**, debemos incluir **axis = 'columns'**.

IMPUTACIÓN DE DATOS FALTANTES

En lugar de eliminar los registros con datos faltantes y potencialmente descartar otros datos junto con ellos, podríamos **“llenar” los huecos**, reemplazando los valores faltantes por valores posibles o potenciales. Dicho valor de reemplazo podría ser, entre otros:

- Alguna **medida resumen representativa** (segmentada o no según alguna/s variable/s categóricas), como la **media, la mediana o el modo**, calculada sobre los datos existentes
- Un **valor aleatorio** dentro del rango de valores observados
- Un valor estimado a partir de los datos observados utilizando una técnica de **interpolación**

IMPUTACIÓN DE DATOS FALTANTES

Supongamos que tenemos un dataset con datos de hogares de Rosario y existen datos faltantes en la variable **precio_usd**.

```
data = pd.read_excel('datasets/hogares.xlsx')  
  
print(data)
```

	id_propiedad	distrito	barrio	ambientes	precio_usd
0	1	oeste	bella_vista	3	87000.5
1	2	norte	refineria	3	104000.0
2	3	oeste	cinco_esquinas	2	98000.0
3	4	sur	saladillo	2	85000.0
4	5	centro	centro	2	65000.0
5	6	sur	hospitales	3	96000.0
6	7	centro	echesortu	1	36500.0
7	8	oeste	parque	3	120000.5
8	9	norte	alberdi	2	105000.0

IMPUTACIÓN DE DATOS FALTANTES

Hay dos valores faltantes en la columna **precio_usd**.

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 50 entries, 0 to 49
```

```
Data columns (total 5 columns):
```

#	Column	Non-Null Count	Dtype
0	id_propiedad	50 non-null	int64
1	distrito	50 non-null	object
2	barrio	50 non-null	object
3	ambientes	50 non-null	int64
4	precio_usd	48 non-null	float64

IMPUTACIÓN DE DATOS FALTANTES

¿Cuáles son los registros que tienen datos faltantes?

```
print(data.loc[data['precio_usd'].isna()])
```

	id_propiedad	distrito	barrio	ambientes	precio_usd
10	11	centro	martin	2	NaN
13	14	norte	alberdi	2	NaN

Son los registros que poseen los índices 10 y 13.

fillna()

Podríamos imputar los datos faltantes con el **precio promedio** del resto de las propiedades. Para ello contamos con la función **fillna()**.

```
# Creamos una copia del DataFrame
data_mean = data.copy()

# Calculamos el precio promedio
precio_promedio = data_mean['precio_usd'].mean()


# Reemplazamos los valores faltantes con el precio promedio
data_mean['precio_usd'].fillna(precio_promedio, inplace = True)

print(data_mean.iloc[[10, 13]])
```

	id_propiedad	distrito	barrio	ambientes	precio_usd
10	11	centro	martin	2	84555.052083
13	14	norte	alberdi	2	84555.052083

fillna()

Otra elección podría ser imputar los valores faltantes con el precio promedio segmentado **según el barrio en el que se encuentra la propiedad.**

```
# Calculamos los precios promedio de las propiedades según el barrio   
(data.groupby('barrio'))['precio_usd'].mean()
```

```
barrio  
alberdi          128280.100000  
bella_vista      87000.500000  
centro           76372.222222  
cinco_esquinas   67525.000000  
echesortu        36500.000000  
hospitales       90100.000000  
martin           80120.000000  
parque           92180.200000  
...          ...
```

fillna()

Podemos utilizar la información anterior para imputar los valores faltantes de la siguiente manera:

```
# Creamos una copia del DataFrame
data_grouped_mean = data.copy()

# Reemplazamos los valores faltantes con el precio promedio
data_grouped_mean['precio_usd'].fillna((data.groupby('barrio'))['precio_usd'].transform('mean'))

print(data_grouped_mean.iloc[[10,13]])
```

	id_propiedad	distrito	barrio	ambientes	precio_usd
10	11	centro	martin	2	80120.0
13	14	norte	alberdi	2	128280.1

fillna()

Comparemos las dos estrategias utilizadas:

Imputar utilizando un **promedio general**

```
print(data_mean.iloc[[10,13]])
```

	id_propiedad	distrito	barrio	ambientes	precio_usd
10	11	centro	martin	2	84555.052083
13	14	norte	alberdi	2	84555.052083

Imputar utilizando el **promedio segmentado por barrio**

```
print(data_grouped_mean.iloc[[10,13]])
```

	id_propiedad	distrito	barrio	ambientes	precio_usd
10	11	centro	martin	2	80120.0
13	14	norte	alberdi	2	128280.1

IMPUTACIÓN POR INTERPOLACIÓN

La interpolación es una técnica que se utiliza para imputar valores faltantes con la ayuda de valores existentes “vecinos”.

En la práctica, se suele utilizar cuando se trabaja con datos de **series de tiempo**.

Ejemplo

Si estamos trabajando con registros de temperatura a lo largo del día y, por algún motivo, **no contamos con el registro de dicha información a las 13 hs.**, pero sí de las 10 y las 14 hs., podríamos utilizar una interpolación para imputar ese dato faltante.

IMPUTACIÓN POR INTERPOLACIÓN

La **interpolación lineal** es la forma más sencilla de hacer una interpolación. Teniendo dos puntos (x_0, y_0) , (x_1, y_1) podemos calcular una única recta que pase por los mismos. La función obtenida sirve para calcular el valor de y' para cualquier valor de x' perteneciente al intervalo $[x_0, x_1]$.

$$\frac{x_1 - x_0}{x' - x_0} = \frac{y_1 - y_0}{y' - y_0}$$

IMPUTACIÓN POR INTERPOLACIÓN

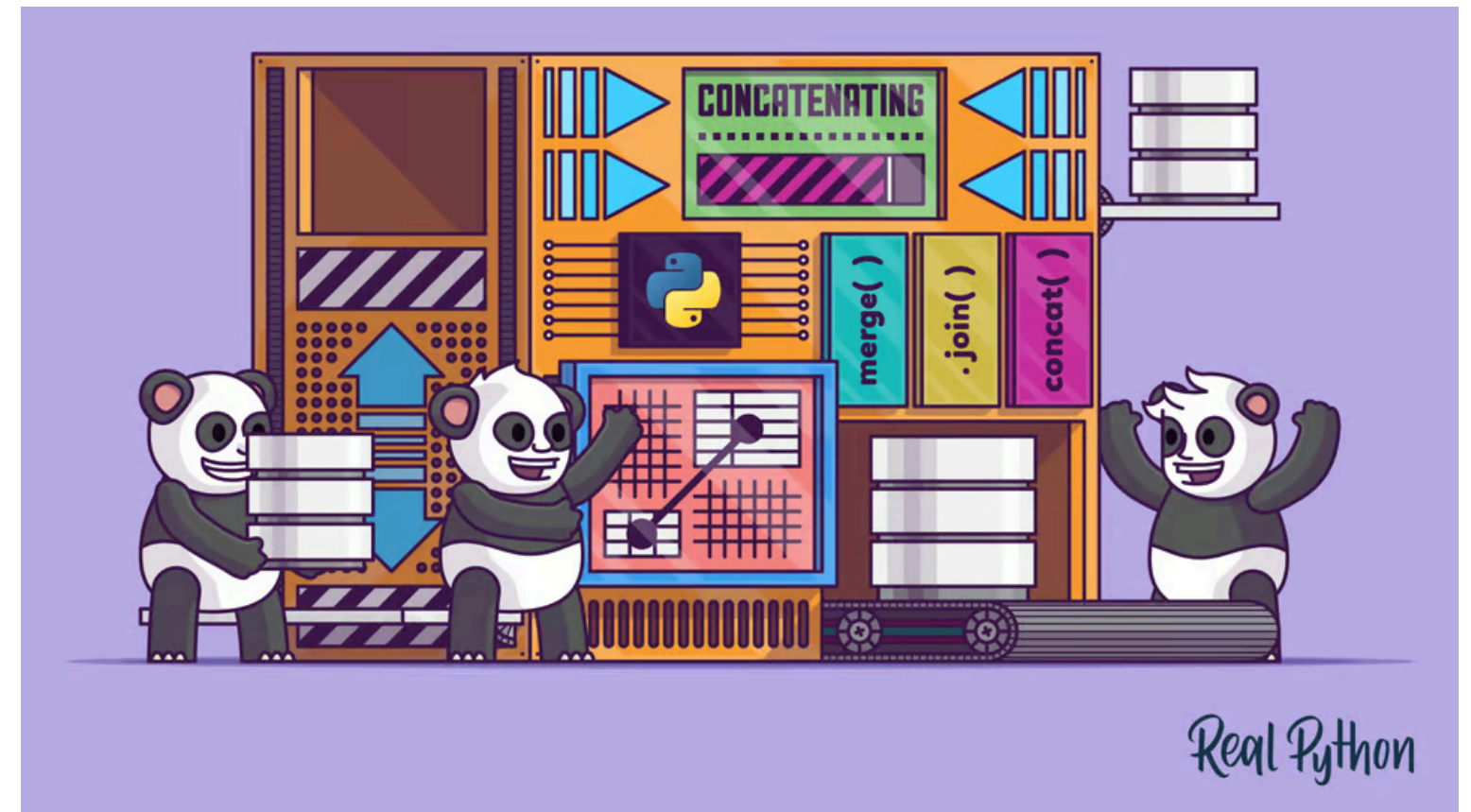
La **interpolación polinómica** es una forma de **interpolación global**. La idea es buscar un polinomio que pase por todos los puntos que tenemos como dato para obtener una ecuación que estime y' como $f(x')$.

COMBINACIONES DE CONJUNTOS DE DATOS

Son operaciones que se realizan entre diferentes datasets para ampliar la información disponible para el análisis.

PANDAS ofrece varios métodos para combinar DataFrames. Los más comunes son:

- `merge()`
- `concat()`
- `join()`



concat()

Se utiliza para concatenar dos o más DataFrames **a lo largo de un eje específico**, ya sea horizontal o verticalmente, información que se especifica en el argumento **axis**.

Por ejemplo:

```
# Definimos dos DataFrames de ejemplo
df1 = pd.DataFrame({'A': [1,2,3], 'B': [4,5,6]}, index = [0,1,2])

df2 = pd.DataFrame({'A': [4,5,6], 'B': [7,8,9], 'C': [10,11,12]}, index = [0,1,2])

# Concatenar los DataFrames verticalmente
nuevo_df = pd.concat([df1, df2], axis = 0)
```

concat()

```
# Imprimimos el nuevo DataFrame
print(nuevo_df)
```

	A	B	C
0	1	4	NaN
1	2	5	NaN
2	3	6	NaN
1	4	7	10.0
2	5	8	11.0
3	6	9	12.0

concat()

Por *default*, el tipo de join es un **outer join**, pero se puede cambiar por **inner** seteando el argumento **join**. ¿Qué cambia en este caso?

```
# Volvemos a concatenar verticalmente, pero seteando join = 'inner'
nuevo_df = pd.concat([df1, df2], axis = 0, join = 'inner')

# Imprimimos el nuevo DataFrame
print(nuevo_df)
```

	A	B
0	1	4
1	2	5
2	3	6
1	4	7
2	5	8
3	6	9

concat()

Si concatenamos los mismos DataFrames `df1` y `df2` **horizontalmente** (**`axis = 1`**) el resultado es:

```
# Concatenar los DataFrames horizontalmente
nuevo_df = pd.concat([df1, df2], axis = 1)

# Imprimir el nuevo DataFrame
print(nuevo_df)
```

	A	B	A	B	C
0	1.0	4.0	NaN	NaN	NaN
1	2.0	5.0	4.0	7.0	10.0
2	3.0	6.0	5.0	8.0	11.0
3	NaN	NaN	6.0	9.0	12.0

concat()

¿Y si seteamos **join = 'inner'**?

```
# Concatenar los DataFrames horizontalmente, pero seteando join = 'inner'  
nuevo_df = pd.concat([df1, df2], axis = 1, join = 'inner')  
  
# Imprimir el nuevo DataFrame  
print(nuevo_df)
```

	A	B	A	B	C
1	2	5	4	7	10
2	3	6	5	8	11

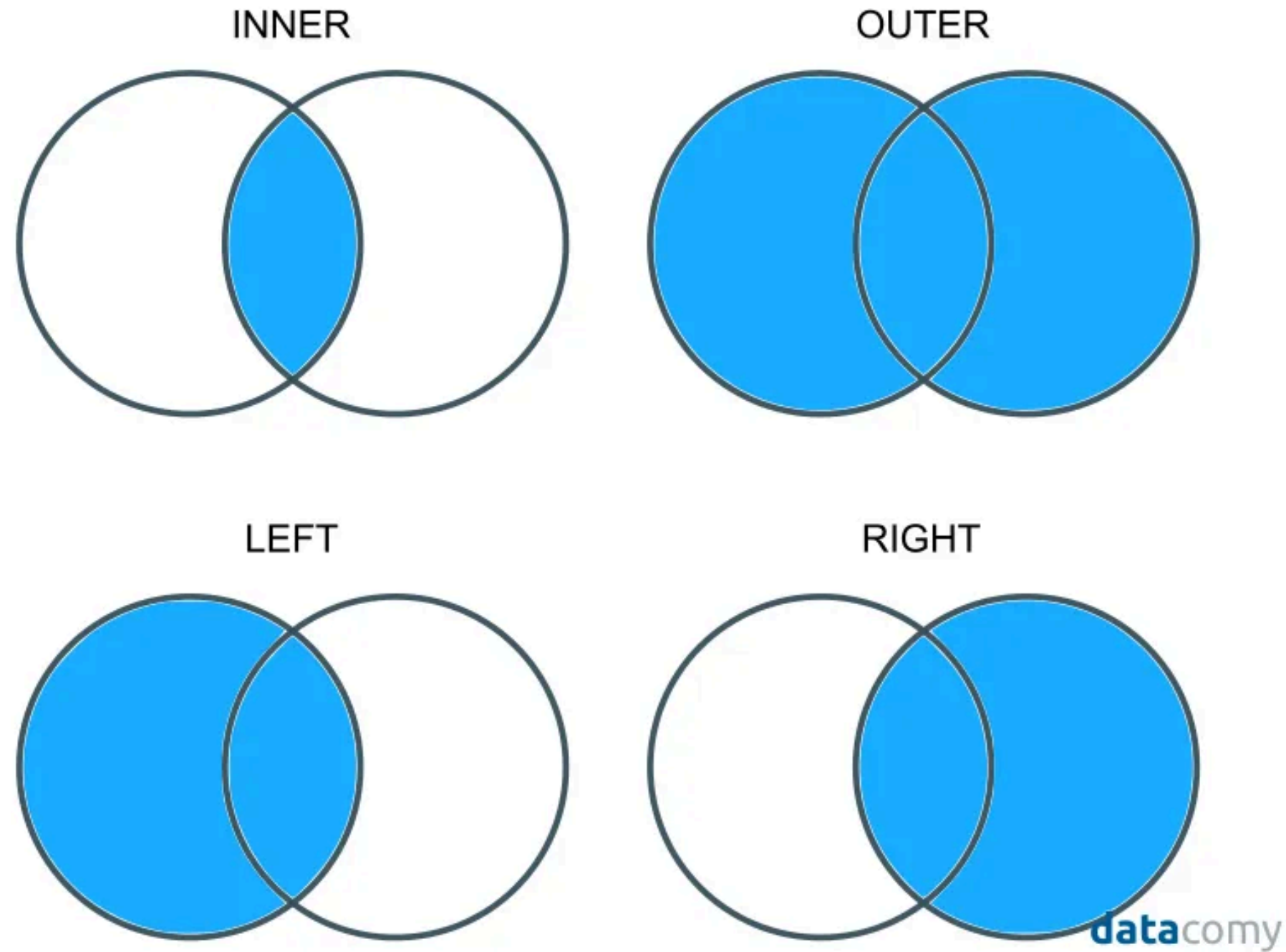
merge()

El método **merge()** es la opción más común y flexible para la combinación de DataFrames, permitiendo unir dos o más datasets en base a **una o más columnas que funcionan como *keys***.

Este tipo de operaciones son particularmente importantes en bases de datos relacionales (por ejemplo, aquellas basadas en SQL).

merge() - Tipos de uniones

Puede elegirse entre las siguientes opciones para el parámetro **how**:



merge() - Otros parámetros importantes

- **left/right**: DataFrames que se van a combinar en el lado izquierdo/derecho.
- **on**: nombre/s de **la/s columna/s para realizar la unión**, la/s cual/es **debe/n ser compartida/s por ambos datasets**. Si no se especifica este argumento, se unirá de forma predeterminada utilizando los índices.
- **left_on**: columna/s en el DataFrame izquierdo (**left**) para usar como *key/s*. Puede ser el nombre de una única columna o una lista de los nombres de varias columnas.
- **right_on**: análogo a **left_on** para el DataFrame derecho (**right**).

merge()

Veamos cómo usar esta función en el marco de un ejemplo.

Como resultado de una encuesta de hogares obtenemos una tabla del hogar con las variables `id_hogar` y `barrio` (**Tabla hogares**) y una tabla de personas con las variables `id_persona`, `motivo_viaje`, `genero` e `id_hogar` (**Tabla personas**).

merge()

Tabla personas

id_persona	motivo_viaje	genero	id_hogar
3449	trabajo	femenino	450956
3450	no_trabajo	masculino	450956
3451	trabajo	masculino	450958

Tabla hogares

id_hogar	barrio
450956	Centro
450957	Belgrano
450958	Lourdes

merge()

```
# Creamos ambas tablas
tabla_hogares = pd.DataFrame({'id_hogar' : ['450956', '450957', '450958'],
                              'barrio' : ['Centro', 'Belgrano', 'Lourdes']})

tabla_personas = pd.DataFrame({'id_persona' : ['3449', '3450', '3451'],
                              'motivo_viaje' : ['trabajo', 'no_trabajo', 'trabajo'],
                              'genero' : ['femenino', 'masculino', 'masculino'],
                              'id_hogar' : ['450956', '450956', '450958']})
```



merge()

Supongamos que queremos calcular la **cantidad de viajes de trabajo por barrio**. Para eso, necesitamos saber a qué barrio pertenece cada persona, lo que podemos averiguar utilizando el **id_hogar** que se encuentra en ambas tablas.

Tendríamos que lograr tener algo así:

id_persona	motivo_viaje	género	id_hogar	barrio
3449	trabajo	femenino	450956	Centro
3450	no_trabajo	masculino	450956	Centro
3451	trabajo	masculino	450958	Lourdes

merge()



UNIONES CRUZADAS (*Cross Joins*)

El método **merge()** presentado anteriormente también permite realizar una **unión cruzada** (*cross join*), la cual devuelve todas las combinaciones posibles entre los registros de dos DataFrames, independientemente de si los valores coinciden o no.

Para realizar una unión cruzada, se debe establecer el parámetro **how** en **how = 'cross'**.

UNIONES CRUZADAS (*Cross Joins*)

```
# Definimos dos DataFrames de ejemplo
df1 = pd.DataFrame({'A': [1, 2]})
df2 = pd.DataFrame({'B': ['a', 'b', 'c']})

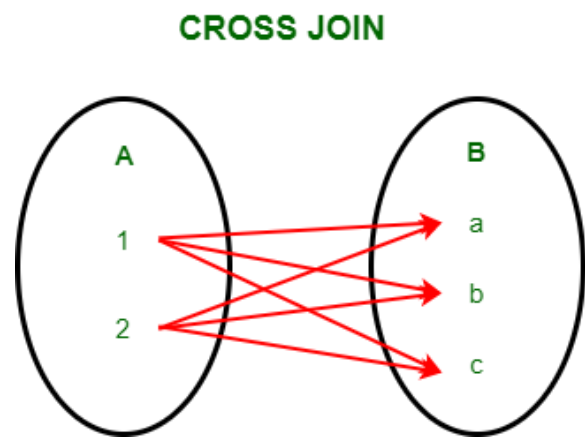
# Realizamos una unión cruzada entre los DataFrames
nuevo_df = pd.merge(df1, df2, how = 'cross')

# Imprimir el nuevo DataFrame resultante
print(nuevo_df)
```

	A	B
0	1	a
1	1	b
2	1	c
3	2	a
4	2	b
5	2	c

¿Cuál es el resultado de la unión cruzada entre **df1** y **df2**?

UNIONES CRUZADAS (*Cross Joins*)



El DataFrame resultante contiene **todas las combinaciones posibles entre los valores de ambas tablas, sin importar si los valores coinciden o no.**

Es importante tener en cuenta que realizar una unión cruzada puede generar un DataFrame muy grande si los DataFrames originales son grandes. Por lo tanto, se debe tener cuidado al utilizar esta técnica y asegurarse de que sea realmente necesaria para el análisis que se está realizando.

join()

La función **join()** se utiliza para combinar dos o más DataFrames **en función de sus índices o columnas**.

Por ejemplo:

```
# Definimos dos DataFrames de ejemplo
df1 = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [4, 5, 6, 7]}, index = ['a',
df2 = pd.DataFrame({'C': [7, 8, 9], 'D': [10, 11, 12]}, index = ['a', 'b

# Unimos los dos DataFrames usando la función join()
df = df1.join(df2)
```

join()

```
# Imprimimos el DataFrame resultante  
print(df)
```

	A	B	C	D
a	1	4	7.0	10.0
b	2	5	8.0	11.0
c	3	6	9.0	12.0
d	4	7	NaN	NaN

Por *default*, se realiza una unión de tipo **left**, aunque puede modificarse a través del parámetro **how**.

join()

También se puede utilizar la función **join()** para combinar DataFrames **en función de columnas específicas**. En este caso, se especifica la columna en la que se desea realizar la unión utilizando el parámetro **on**.

Por ejemplo:

```
# Definimos un nuevo DataFrame
df3 = pd.DataFrame({'A': [1, 2, 3, 4], 'E': [5, 6, 7, 8],
                    'F': [9, 10, 11, 12]}, index = ['a', 'b', 'c', 'd'])

# Unimos df1 y df3 por la columna 'A'
df = df1.join(df3.set_index('A'), on = 'A')
```

join()

El resultado es:

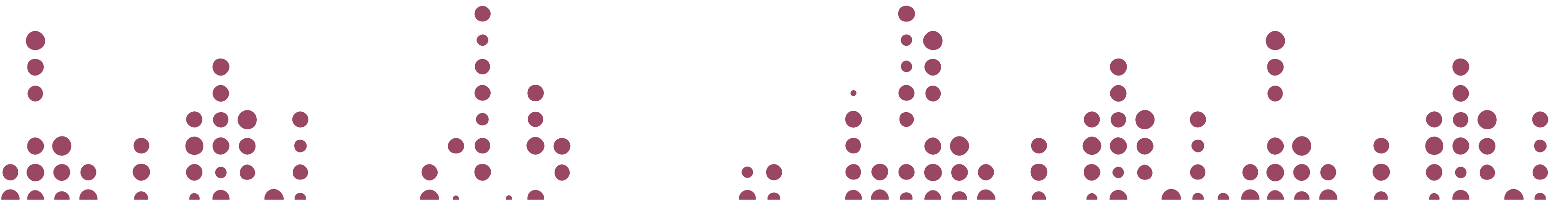
```
# Imprimimos el DataFrame resultante  
print(df)
```

	A	B	E	F
a	1	4	5	9
b	2	5	6	10
c	3	6	7	11
d	4	7	8	12

EXPRESIONES REGULARES

Las expresiones regulares proporcionan una manera flexible de **buscar o hacer coincidir patrones de cadenas en un texto.**

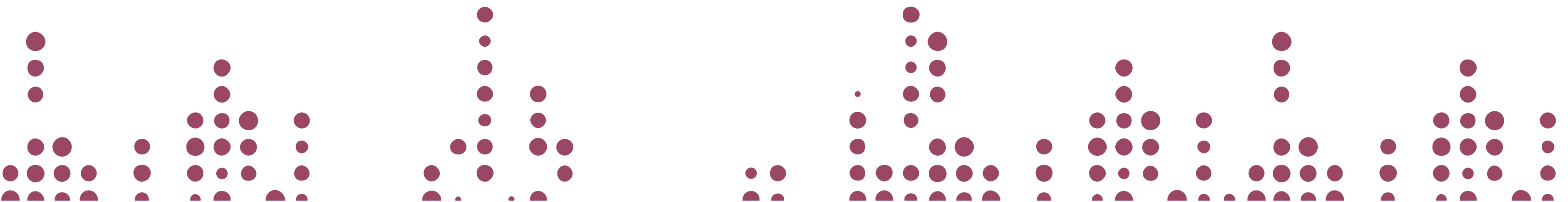
REGEX Es una cadena formada según el lenguaje de expresiones regulares que especifica un patrón de búsqueda determinado.



EXPRESIONES REGULARES EN PYTHON

Python cuenta con un paquete llamado **re** que incluye un conjunto de funciones para el trabajo con expresiones regulares y que pueden agruparse dentro de tres categorías diferentes: **coincidencia de patrones**, **sustitución** y **división**, aunque naturalmente están todas relacionadas.

re.search(pattern, string, flags = 0) Busca la primera ocurrencia del patrón **pattern** en la cadena **string** y devuelve un objeto del tipo match.



EXPRESIONES REGULARES EN PYTHON

`re.split(pattern, string, maxsplit = 0, flags = 0)`

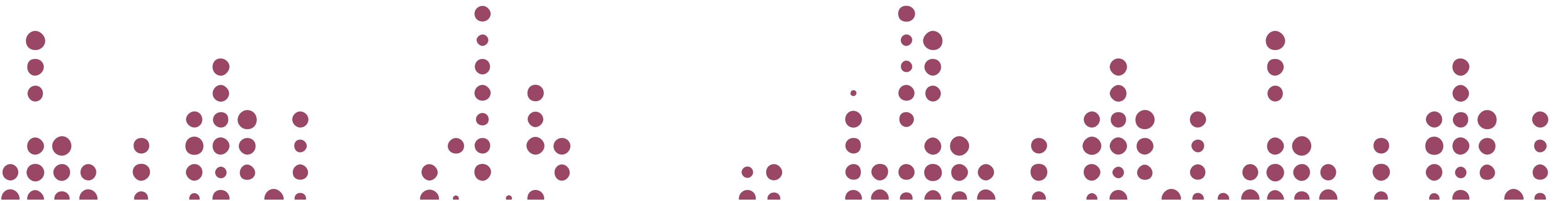
Parte la cadena en cada ocurrencia del patrón. Si el patrón está entre paréntesis, también se devuelve el texto en cada ocurrencia.

Si `maxsplit` se configura como un valor diferente a cero, entonces, se devuelven como máximo ese número de *maxsplits* y el resto se devuelve como una cadena.



EXPRESIONES REGULARES EN PYTHON

`re.findall(pattern, string, flags = 0)` Devuelve todas las ocurrencias del patrón encontradas en la cadena como una lista o una tupla.



EXPRESIONES REGULARES EN PYTHON

`re.sub(pattern, repl, string, count = 0, flags = 0)` Se utiliza para realizar reemplazos en una cadena. Devuelve una cadena en la cual se produjo el reemplazo de cada ocurrencia del patrón `pattern` por el valor `repl`. Si no se encontró ningún valor, entonces se devuelve la cadena original sin modificar.



EXPRESIONES REGULARES EN PYTHON

¿Qué puede ser una *regex*?

- **Caracter.** Excepto los especiales, todos los caracteres coinciden consigo mismos. Por ejemplo: la letra **a**.
- **Secuencia de caracteres.** Por ejemplo, la palabra **casa**.
- **Caracteres especiales.** Por ejemplo, **\d** matchea con cualquier dígito (0-9) y **\d{4}** con cualquier secuencia de 4 dígitos. *Ver cuadro completo de caracteres especiales en el material de estudio.*



EXPRESIONES REGULARES EN PYTHON

Los caracteres especiales pueden combinarse para dar lugar a expresiones regulares que nos posibiliten la búsqueda de patrones más complejos.

Por ejemplo, la siguiente expresión regular permite buscar fechas en el formato **DD/MM/YYYY** que correspondan al **mes de mayo**:

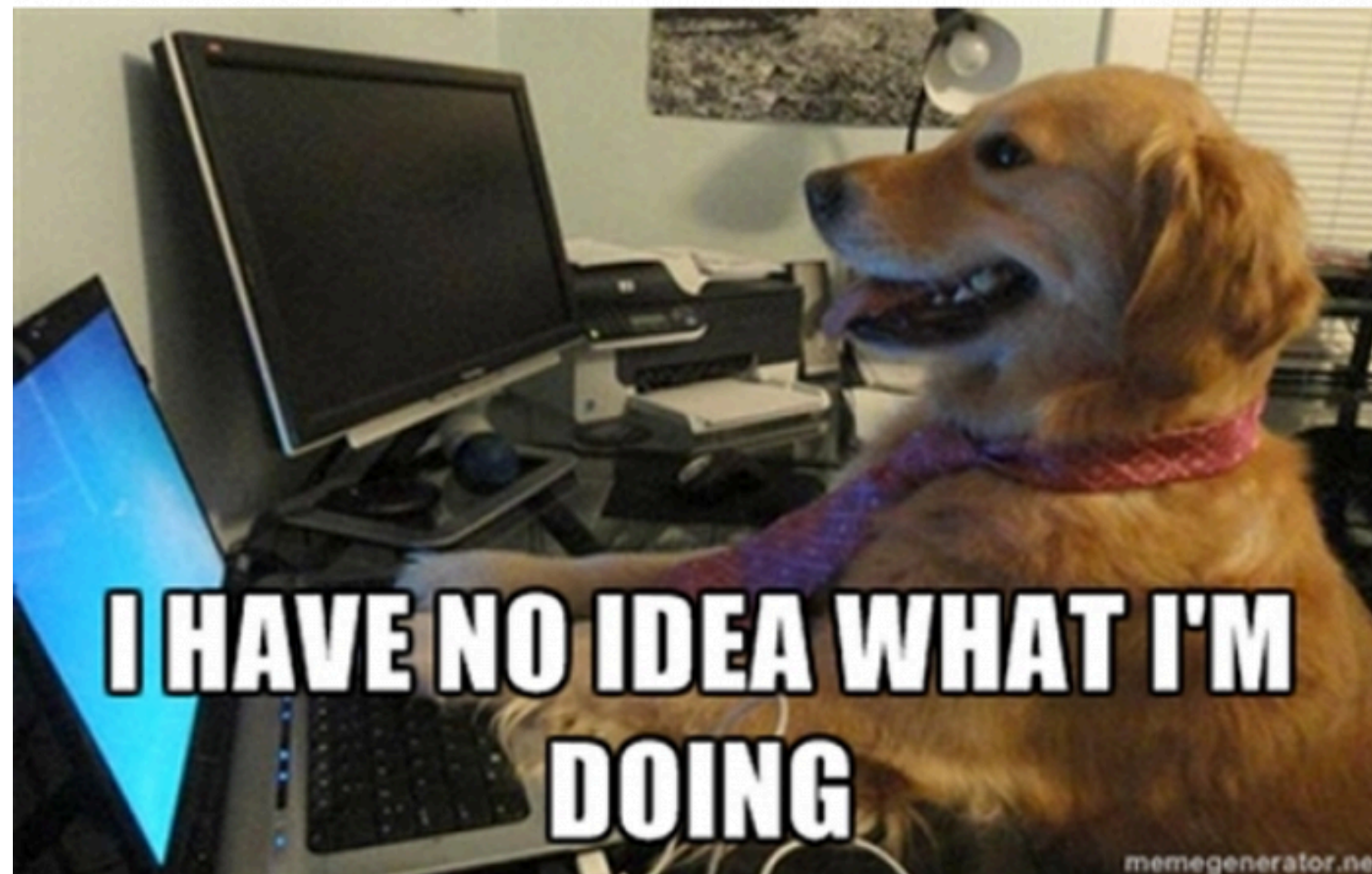
`\b(?:0[1-9] | [12][0-9] | 3[01])\s/05\s/\d{4}\b`.



EXPRESIONES REGULARES EN PYTHON

`\b(?:0[1-9]|[12][0-9]|3[01])\s05\s\d{4}\b`

```
output = new KeyValuePair<string, string>();  
string pattern = @"([A-Za-z0-9._-])+([0-9[\\]])*\={1}([A-Za-z0-9._-])+";
```



EXPRESIONES REGULARES EN PYTHON

Veamos un ejemplo. Supongamos que tenemos el siguiente objeto string: 'Se necesitan 30 azulejos para revestir 1 m2'. La expresión regular `\D+` coincidirá con una o más ocurrencias (+) de todo lo que no sea un dígito (`\D`).

```
import re

re.search(r'\D+', 'Se necesitan 30 azulejos para revestir 1 m2')
```

```
<re.Match object; span=(0, 13), match='Se necesitan '>
```

¿Qué nos devuelve la función `re.search()`?

EXPRESIONES REGULARES EN PYTHON

¿Cuál es la diferencia con `re.findall()`?

```
re.findall(r'\D+', 'Se necesitan 30 azulejos para revestir 1 m2')  
['Se necesitan ', ' azulejos para revestir ', ' m']
```


EXPRESIONES REGULARES EN PYTHON

Si en realidad fueran 15 azulejos los requeridos para revestir 1 m2, puedo utilizar la función **re.sub()** para hacer el reemplazo correspondiente:

```
re.sub(r'30', '15', 'Se necesitan 30 azulejos para revestir 1 m2')
```

```
'Se necesitan 15 azulejos para revestir 1 m2'
```

`str.extract()`

El método `str.extract()` de `pandas` permite realizar la extracción de subcadenas de una columna de un DataFrame que coincidan con un patrón determinado.

Supongamos que tenemos este mini-DataFrame:

```
precios_deptos = pd.DataFrame({'id' : [1,2,3], 'precio' : ['USD 87000',  
  
print(precios_deptos)
```

	id	precio
0	1	USD 87000
1	2	usd 104000
2	3	USD 95000

str.extract()

Podemos utilizar esta operación para extraer el precio en otra columna (y así poder trabajar luego con esa información, separada de la moneda).

```
precios_deptos['precio_usd'] = precios_deptos['precio'].str.extract(r'(\d+)')  
  
print(precios_deptos)
```

	id	precio	precio_usd
0	1	USD 87000	87000
1	2	usd 104000	104000
2	3	USD 95000	95000

A través de la regex **(\d+)** se busca y captura una secuencia de uno o más dígitos consecutivos en una cadena.

`str.extract()`

Si existe más de un *match* de la regex, `str.extract()` extraerá únicamente la **primera coincidencia**.

```
precios_deptos = pd.DataFrame({'id' : [1,2,3], 'precio' : ['USD 87000 EUR 78577', 'usd 104000 eur 93931', 'USD 95000 EUR 85803']})  
  
precios_deptos['precio_usd'] = precios_deptos['precio'].str.extract(r'(\d+)')  
  
print(precios_deptos)
```

	id	precio	precio_usd
0	1	USD 87000 EUR 78577	87000
1	2	usd 104000 eur 93931	104000
2	3	USD 95000 EUR 85803	95000

Para otros casos, existe la variante `str.extractall()`.