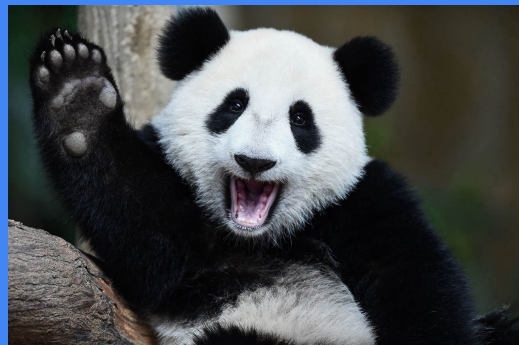
Programación para el análisis de datos

Federico Pousa
fpousa@udesa.edu.ar

Hoy:
Pandas



¿Qué es? ¿Por qué usarlo?

Pandas es una librería muy utilizada en python para el análisis de datos tabulares

- Manipulación de datos estructurados/tablas
  - Tablas de excel
  - csv(s)
  - SQL
  - R data frames
  - Etc

- Nos permite hacer un montón de cosas con los datos:
  - Importarlos
  - Limpiarlos
  - Explorarlos
  - Procesarlos

Nos va a permitir luego interactuar con otras librerías para hacer Machine Learning y estadística
Referencia: https://pandas.pydata.org/

Estructuras: Series

Las series son arreglos unidimensionales que pueden guardar datos de cualquier tipo (números, palabras, objetos, etc)

Las series tienen un campo especial que es el index, que son los ejes por lo que ordena la información

La sintaxis canonica es:

```
s = pd.Series(data, index=index)
```

```
In [1]: import pandas as pd

In [2]: a_series = pd.Series([31,28,31,30,31,30],
                             index=['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio'])

In [3]: a_series

Out[3]: Enero      31
        Febrero    28
        Marzo      31
        Abril      30
        Mayo       31
        Junio      30
        dtype: int64

In [4]: same_series = pd.Series({'Enero':31,'Febrero':28,'Marzo':31,'Abril':30, 'Mayo':31, 'Junio':30})

In [5]: same_series

Out[5]: Enero      31
        Febrero    28
        Marzo      31
        Abril      30
        Mayo       31
        Junio      30
        dtype: int64
```

Estructuras: Dataframes

Los dataframes son como tablas de Excel, donde cada columna es una serie.

Los dataframe pueden ser creados con diferentes tipos de datos

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

La sintaxis canonica es:

```
s = pd.DataFrame(data)
```

También se le puede pasar el index y un campo nuevo llamado column que es una lista del nombre de cada columna

```
In [6]: data = [['Enero', 31, 'E'],
                ['Febrero', 28, 'F'],
                ['Marzo', 31, 'M'],
                ['Abril', 30, 'A'],
                ['Mayo', 31, 'M'],
                ['Junio', 30, 'J']]

        df = pd.DataFrame(data, columns = ['Mes', 'Dias', 'Inicial'])
        df
```

Out[6]:

	Mes	Dias	Inicial
0	Enero	31	E
1	Febrero	28	F
2	Marzo	31	M
3	Abril	30	A
4	Mayo	31	M
5	Junio	30	J

Manejo básico

Ya tenemos forma de crear un DataFrame, sobre este tipo de datos podemos hacer algunas operaciones básicas:

- Pedir el atributo .columns para ver las columnas del DataFrame.
- Pedir el atributo .values para ver los valores del DataFrame.
- Llamar al método .info() para tener información básica del objeto.
- Llamar al método .describe() para tener información de la distribución en las columnas numéricas.
- Llamar al método .head() para mirar las primeras 5 filas del DataFrame.
- Llamar al método .tail() para mirar las últimas 5 filas del DataFrame.
- Usar .NombreDeColumna o df[“NombreDeColumna”] para acceder a una única columna.
- Usar df[start:end] para acceder a las filas [start,end).

Manejo básico

```
df = pd.DataFrame(data, columns = ['Mes', 'Dias', 'Inicial'])
df
```

Out[6]:

	Mes	Dias	Inicial
0	Enero	31	E
1	Febrero	28	F
2	Marzo	31	M
3	Abril	30	A
4	Mayo	31	M
5	Junio	30	J

In [7]:
```
df.columns
```

Out[7]:
```
Index(['Mes', 'Dias', 'Inicial'], dtype='object')
```

In [8]:
```
df.values
```

Out[8]:
```
array([['Enero', 31, 'E'],
       ['Febrero', 28, 'F'],
       ['Marzo', 31, 'M'],
       ['Abril', 30, 'A'],
       ['Mayo', 31, 'M'],
       ['Junio', 30, 'J']], dtype=object)
```

Manejo básico

```
In [9]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Mes      6 non-null      object
 1   Dias     6 non-null      int64
 2   Inicial  6 non-null      object
dtypes: int64(1), object(2)
memory usage: 272.0+ bytes
```

```
In [10]: df.describe()
```

Out[10]:

	Dias
count	6.000000
mean	30.166667
std	1.169045
min	28.000000
25%	30.000000
50%	30.500000
75%	31.000000
max	31.000000

Manejo básico

```
In [11]: df.head()
```

Out[11]:

	Mes	Dias	Inicial
0	Enero	31	E
1	Febrero	28	F
2	Marzo	31	M
3	Abril	30	A
4	Mayo	31	M

```
In [12]: df.tail()
```

Out[12]:

	Mes	Dias	Inicial
1	Febrero	28	F
2	Marzo	31	M
3	Abril	30	A
4	Mayo	31	M
5	Junio	30	J

```
In [13]: df.Dias
```

```
Out[13]: 0    31
         1    28
         2    31
         3    30
         4    31
         5    30
         Name: Dias, dtype: int64
```

```
In [14]: df['Dias']
```

```
Out[14]: 0    31
         1    28
         2    31
         3    30
         4    31
         5    30
         Name: Dias, dtype: int64
```

```
In [15]: df[1:3]
```

Out[15]:

	Mes	Dias	Inicial
1	Febrero	28	F
2	Marzo	31	M

Lectura de DataFrame

Más allá de la variedad de constructores, muchas veces vamos a directamente leer un DataFrame entero a partir de otra fuente de información.

Pandas permite cargar rápidamente información almacenada en diferentes formatos:

- csv
- tsv
- excel
- json
- html
- zip

Por ejemplo, para leer un csv vamos a utilizar la función read_csv() que admite diversos parámetros para cambiar el delimitador y muchas otras opciones. En el uso de la función debemos indicar el *path* del csv que queremos leer, que perfectamente puede ser una url en la web.

Lectura de DataFrame

```
In [16]: df_crime = pd.read_csv('crime.csv', encoding="ISO-8859-1")
```

```
In [17]: df_crime.head()
```

Out[17]:

	INCIDENT_NUMBER	OFFENSE_CODE	OFFENSE_CODE_GROUP	OFFENSE_DESCRIPTION	DISTRICT	REPORTING_AREA	SHOOTING	OCCURRED_ON_DAT
0	I182070945	619	Larceny	LARCENY ALL OTHERS	D14	808	NaN	2018-09-02 13:00:0
1	I182070943	1402	Vandalism	VANDALISM	C11	347	NaN	2018-08-21 00:00:0
2	I182070941	3410	Towed	TOWED MOTOR VEHICLE	D4	151	NaN	2018-09-03 19:27:0
3	I182070940	3114	Investigate Property	INVESTIGATE PROPERTY	D4	272	NaN	2018-09-03 21:16:0
4	I182070938	3114	Investigate Property	INVESTIGATE PROPERTY	B3	421	NaN	2018-09-03 21:05:0

```
In [18]: df_crime[2:5].to_csv('nuevoarchivo.csv')
```

```
In [19]: df.shape
```

```
Out[19]: (6, 3)
```

```
In [20]: len(df)
```

```
Out[20]: 6
```

Acceso avanzado

Vimos cómo acceder a una columna o a una cierta cantidad de filas, pero nos va a interesar acceder a recortes más específicos: conjuntos de columnas, filas que cumplan ciertas condiciones, etc.

El acceso a más de una columna se realiza mediante *df[lista_de_columnas]*:

```
In [21]: df_crime[['OFFENSE_CODE', 'OFFENSE_DESCRIPTION']]
```

Out[21]:

	OFFENSE_CODE	OFFENSE_DESCRIPTION
0	619	LARCENY ALL OTHERS
1	1402	VANDALISM
2	3410	TOWED MOTOR VEHICLE
3	3114	INVESTIGATE PROPERTY
4	3114	INVESTIGATE PROPERTY
...	...	...
319068	3125	WARRANT ARREST
319069	111	MURDER, NON-NEGLIGIENT MANSLAUGHTER
319070	3125	WARRANT ARREST
319071	3125	WARRANT ARREST
319072	3125	WARRANT ARREST

319073 rows × 2 columns

Acceso avanzado

Si queremos acceder a ciertas filas, podemos pasar una lista de booleanos que nos diga con qué filas queremos quedarnos y con cuáles no:

In [22]:
```
rows_to_keep = [False]*df_crime.shape[0]
rows_to_keep[5] = True
rows_to_keep[14] = True
df_crime[rows_to_keep]
```

Out[22]:

	INCIDENT_NUMBER	OFFENSE_CODE	OFFENSE_CODE_GROUP	OFFENSE_DESCRIPTION	DISTRICT	REPORTING_AREA	SHOOTING	OCCURRED_ON_DA
5	I182070936	3820	Motor Vehicle Accident Response	M/V ACCIDENT INVOLVING PEDESTRIAN - INJURY	C11	398	NaN	2018-09-03 21:09:
14	I182070921	3201	Property Lost	PROPERTY - LOST	B3	469	NaN	2018-09-02 14:00:

Acceso avanzado

A nadie le sirve armar una lista de booleanos constructivamente, pero esto nos sirve para devolver las filas que cumplan una condición.

```
In [23]: df_crime['OFFENSE_DESCRIPTION'].str.contains('MURDER')
Out[23]: 0         False
         1         False
         2         False
         3         False
         4         False
                   ...
         319068    False
         319069     True
         319070    False
         319071    False
         319072    False
         Name: OFFENSE_DESCRIPTION, Length: 319073, dtype: bool
```

```
In [24]: df_crime[df_crime['OFFENSE_DESCRIPTION'].str.contains('MURDER')]
```

Out[24]:

	INCIDENT_NUMBER	OFFENSE_CODE	OFFENSE_CODE_GROUP	OFFENSE_DESCRIPTION	DISTRICT	R
3259	I182067317	111	Homicide	MURDER, NON-NEGLIGIENT MANSLAUGHTER	B2	
5689	I182064699	111	Homicide	MURDER, NON-NEGLIGIENT MANSLAUGHTER	B3	
6278	I182064056	111	Homicide	MURDER, NON-NEGLIGIENT MANSLAUGHTER	B3	
10813	I182059055	111	Homicide	MURDER, NON-NEGLIGIENT MANSLAUGHTER	B3	
11021	I182058835	111	Homicide	MURDER, NON-NEGLIGIENT MANSLAUGHTER	B3	
...	...	...	...	...	...	

Acceso avanzado

La condición puede ser más elaborada, mezclando diferentes Series.

```
In [25]: df_crime[(df_crime['OFFENSE_DESCRIPTION'].str.contains('MURDER')) | (df_crime['OFFENSE_CODE']==619)]
```

Out[25]:

	INCIDENT_NUMBER	OFFENSE_CODE	OFFENSE_CODE_GROUP	OFFENSE_DESCRIPTION	DISTRICT	REPORTING_AREA	SHOOTING	C
0	I182070945	619	Larceny	LARCENY ALL OTHERS	D14	808	NaN	
45	I182070885	619	Larceny	LARCENY ALL OTHERS	B3	456	NaN	
110	I182070816	619	Larceny	LARCENY ALL OTHERS	C6	185	NaN	
150	I182070777	619	Larceny	LARCENY ALL OTHERS	C11	388	NaN	
205	I182070707	619	Larceny	LARCENY ALL OTHERS	D14	782	NaN	
...	...	...	...	...	...	...	...	
318983	I130304171-00	619	Larceny	LARCENY OTHER $200 & OVER	B3	428	NaN	
318993	I130194606-00	619	Larceny	LARCENY OTHER $200 & OVER	D4	136	NaN	
319026	I120283195-00	619	Larceny	LARCENY ALL OTHERS	B2	911	NaN	
319048	I110261417-00	619	Larceny	LARCENY OTHER $200 & OVER	B2	324	NaN	
319069	I030217815-08	111	Homicide	MURDER, NON-NEGLIGIENT MANSLAUGHTER	E18	520	NaN	

Acceso avanzado

Más allá de los accesos vistos con el operador [], existen las funciones *loc* e *iloc* que sirven para indexación general.

loc sirve para indexar tanto filas cómo columnas utilizando sus valores. Para las filas se utiliza el valor del índice (que no necesariamente es numérico) y para las columnas los nombres. Adicionalmente, se puede usar una lista de booleanos para las filas.

```
In [26]: df_crime.loc[205, 'OFFENSE_CODE']
Out[26]: 619

In [27]: df_crime.loc[205, ['OFFENSE_CODE', 'DISTRICT']]
Out[27]: OFFENSE_CODE    619
         DISTRICT        D14
         Name: 205, dtype: object

In [28]: df_crime.loc[205:207, ['OFFENSE_CODE', 'DISTRICT']]
Out[28]:
```

	OFFENSE_CODE	DISTRICT
205	619	D14
206	520	C11
207	2007	C11

```
In [29]: df_crime.loc[(df_crime['OFFENSE_CODE']==619), ['OFFENSE_CODE', 'DISTRICT']]
Out[29]:
```

	OFFENSE_CODE	DISTRICT
0	619	D14
45	619	B3
110	619	C6

Acceso avanzado

iloc se utiliza para acceder directamente con los rangos numéricos de las filas y columnas buscadas.

```
In [30]: df_crime.iloc[3]
Out[30]: INCIDENT_NUMBER                    I182070940
         OFFENSE_CODE                             3114
         OFFENSE_CODE_GROUP          Investigate Property
         OFFENSE_DESCRIPTION         INVESTIGATE PROPERTY
         DISTRICT                                  D4
         REPORTING_AREA                           272
         SHOOTING                                 NaN
         OCCURRED_ON_DATE         2018-09-03 21:16:00
         YEAR                                    2018
         MONTH                                      9
         DAY_OF_WEEK                           Monday
         HOUR                                      21
         UCR_PART                          Part Three
         STREET                            NEWCOMB ST
         Lat                                42.334182
         Long                              -71.078664
         Location              (42.33418175, -71.07866441)
         Name: 3, dtype: object
```

```
In [31]: df_crime.iloc[3:5]
```

Out[31]:

	INCIDENT_NUMBER	OFFENSE_CODE	OFFENSE_CODE_GROUP	OFFEI
3	I182070940	3114	Investigate Property	
4	I182070938	3114	Investigate Property	

```
In [32]: df_crime.iloc[3:5, 1:3]
```

Out[32]:

	OFFENSE_CODE	OFFENSE_CODE_GROUP
3	3114	Investigate Property
4	3114	Investigate Property

Cambiando la información

Los accesos que vimos sirven también para crear y modificar información al usarlos en una asignación.

```
In [33]: df_crime['NEW_COL'] = 1

In [34]: df_crime.loc[:, 'NEW_COL_2'] = 2

In [35]: df_crime.loc[0,'NEW_COL'] = 3

In [36]: df_crime
```

Out[36]:

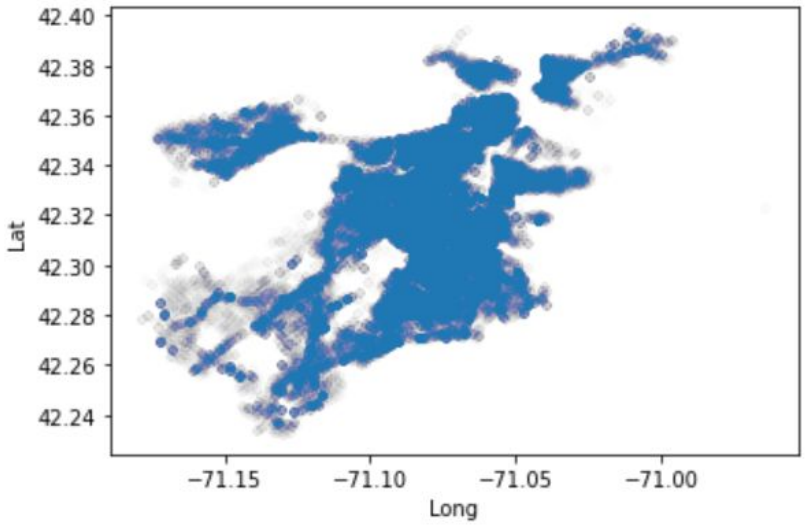
NG	OCCURRED_ON_DATE	YEAR	MONTH	DAY_OF_WEEK	HOUR	UCR_PART	STREET	Lat	Long	Location	NEW_COL	NEW_COL_2
laN	2018-09-02 13:00:00	2018	9	Sunday	13	Part One	LINCOLN ST	42.357791	-71.139371	(42.35779134, -71.13937053)	3	2
laN	2018-08-21 00:00:00	2018	8	Tuesday	0	Part Two	HECLA ST	42.306821	-71.060300	(42.30682138, -71.06030035)	1	2
laN	2018-09-03 19:27:00	2018	9	Monday	19	Part Three	CAZENOVE ST	42.346589	-71.072429	(42.34658879, -71.07242943)	1	2

Visualización básica

Pandas nos provee gráficos para visualizar nuestra información. Está desarrollado sobre matplotlib, librería que veremos más en detalle, por lo que las opciones de pandas las usaremos como una opción rápida y preliminar.

```
In [38]: df_crime = df_crime[df_crime['Long'] < -10]
         df_crime.plot.scatter(x='Long', y='Lat', alpha=0.005)

Out[38]: <AxesSubplot:xlabel='Long', ylabel='Lat'>
```



Iteración y apply

En Pandas hay varias maneras para iterar nuestro dataframe. También tenemos manera de aplicar funciones o transformaciones a todas las filas y columnas sin explícitamente iterarlas. Pandas utiliza fuertemente NumPy en su implementación, así que la sugerencia de *tener cuidado* cuando hacemos algo no vectorizado en Pandas sigue aplicando.

Una primera manera para recorrer los DataFrames es utilizando el método *iterrrows*, que precisamente va iterando las filas, devolviendo tanto su índice como la fila entera:

```
In [39]: for index, row in df.iterrows():
             print('Nueva fila:')
             print(index, row['Mes'], row['Inicial'])
```

```
Nueva fila:
0 Enero E
Nueva fila:
1 Febrero F
Nueva fila:
2 Marzo M
Nueva fila:
3 Abril A
Nueva fila:
4 Mayo M
Nueva fila:
5 Junio J
```

Iteración y apply

Otra posible manera es recorriendo el atributo *index*, combinado con un acceso por [] o por *loc*:

```
In [40]: for index in df.index:
             print(df.Mes[index], df.Dias[index])
```

```
Enero 31
Febrero 28
Marzo 31
Abril 30
Mayo 31
Junio 30
```

```
In [41]: for index in df.index:
             print(df.loc[index, ['Mes', 'Dias']])
```

```
Mes     Enero
Dias       31
Name: 0, dtype: object
Mes     Febrero
Dias         28
Name: 1, dtype: object
Mes     Marzo
Dias       31
Name: 2, dtype: object
Mes     Abril
Dias       30
Name: 3, dtype: object
Mes     Mayo
Dias       31
Name: 4, dtype: object
Mes     Junio
Dias       30
Name: 5, dtype: object
```

Iteración y apply

También podemos recorrer el DataFrame utilizando *iloc* y pasando por todo el rango de filas:

```
In [42]: for num_index in range(len(df)):
             print(df.iloc[num_index].Dias, df.iloc[num_index].Mes)
```

```
31 Enero
28 Febrero
31 Marzo
30 Abril
31 Mayo
30 Junio
```

Iteración y apply

En varias situaciones, en vez de iterar el DataFrame, podemos directamente aplicar el cambio que queremos realizar, por ejemplo utilizando un *apply*:

```
In [43]: print(df.apply(lambda row: row.Mes + ' ' + str(row.Dias), axis=1))
```

```
0      Enero 31
1    Febrero 28
2      Marzo 31
3      Abril 30
4       Mayo 31
5      Junio 30
dtype: object
```

```
In [44]: def month_and_days(row):
             return row.Mes + ' ' + str(row.Dias)
         print(df.apply(month_and_days, axis=1))
```

```
0      Enero 31
1    Febrero 28
2      Marzo 31
3      Abril 30
4       Mayo 31
5      Junio 30
dtype: object
```

```
In [45]: df.apply(lambda row: print(row.Mes, row.Dias), axis=1)
```

```
Enero 31
Febrero 28
Marzo 31
Abril 30
Mayo 31
Junio 30
```

```
Out[45]: 0    None
         1    None
         2    None
         3    None
         4    None
         5    None
         dtype: object
```

```
In [46]: df.apply(lambda column: column.to_list()[:len(column.to_list())//2])
```

Out[46]:

	Mes	Dias	Inicial
0	Enero	31	E
1	Febrero	28	F
2	Marzo	31	M

Funciones predefinidas

Hay muchas funciones que aplican sobre el DataFrame y sobre las Series sin necesidad de iterar explícitamente.

```
In [47]: df.sum()
```

```
Out[47]: Mes        EneroFebreroMarzoAbrilMayoJunio
         Dias                                   181
         Inicial                             EFMAMJ
         dtype: object
```

```
In [48]: df.sort_values(by=['Dias']).reset_index(drop=True)
```

Out[48]:

	Mes	Dias	Inicial
0	Febrero	28	F
1	Abril	30	A
2	Junio	30	J
3	Enero	31	E
4	Marzo	31	M
5	Mayo	31	M

Group by

En muchas ocasiones vamos a querer hacer un cálculo o una transformación pero no en todo el DataFrame, sino pensando en *grupos* de filas. El método *groupby()* nos permite agrupar las filas, por ejemplo, por el valor de una columna en particular.
Este método no devuelve un DataFrame, sino que devuelve un *groupby object*, al cual le podremos pedir algunas funciones de agregación y así luego volver a generar otros DataFrames.

```
In [49]: df.groupby('Dias')
Out[49]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7ffa52044be0>

In [50]: df.groupby('Dias')['Mes']
Out[50]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x7ffa52037970>

In [51]: df.groupby('Dias').first()
```

Out[51]:

Dias	Mes	Inicial
28	Febrero	F
30	Abril	A
31	Enero	E

```
In [52]: df.groupby('Dias').first().reset_index()
```

Out[52]:

	Dias	Mes	Inicial
0	28	Febrero	F
1	30	Abril	A
2	31	Enero	E

```
In [54]: df_crime.groupby([
             'DAY_OF_WEEK', 'HOUR'
         ])['INCIDENT_NUMBER'].count().reset_index()
```

Out[54]:

	DAY_OF_WEEK	HOUR	INCIDENT_NUMBER
0	Friday	0	2031
1	Friday	1	1177
2	Friday	2	867
3	Friday	3	499
4	Friday	4	419
...	...	...	...
163	Wednesday	19	2548
164	Wednesday	20	2129
165	Wednesday	21	1934
166	Wednesday	22	1614
167	Wednesday	23	1218

168 rows × 3 columns

Group by

Si bien existen varios métodos de agregación para ser llamados, lo más genérico es utilizar el método *agg* e indicar explícitamente cómo se quieren agregar las diferentes columnas (incluso con funciones propias).

```
In [53]: df.groupby('Inicial').agg({'Dias':['min', 'mean', 'max'], 'Mes':['first', lambda x : x.to_list()]})
```

Out[53]:

	Dias			Mes	
	min	mean	max	first	<lambda_0>
Inicial					
A	30	30.0	30	Abril	[Abril]
E	31	31.0	31	Enero	[Enero]
F	28	28.0	28	Febrero	[Febrero]
J	30	30.0	30	Junio	[Junio]
M	31	31.0	31	Marzo	[Marzo, Mayo]

Reshapes

En Pandas hay muchas maneras de querer cambiar la forma de un DataFrame. Podemos querer *pivotear* una tabla, podemos querer convertir los valores de las filas en columnas o viceversa, y muchas opciones más.

Algunas funciones que sirven para cambiar la forma de un DataFrame:

- pivot: como una pivot table de excel, se puede indicar cuál es el nuevo índice, columnas y valores.
- stack: convierte columnas en niveles de índice.
- unstack: convierte niveles de índice en columnas.
- melt: como un *unpivot* que pasa de un formato *wide* a uno *long*
- explode: toma una lista de valores y lo convierte en una fila por cada valor.

Pivot

In [55]:
```
df_for_pivot
```

Out[55]:

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

In [57]:
```
df_for_pivot.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
```

Out[57]:

	baz			zoo		
bar	**A**	**B**	**C**	**A**	**B**	**C**
foo						
one	1	2	3	x	y	z
two	4	5	6	q	w	t

Explode

In [57]:
```
df_for_explote = pd.DataFrame(
                    list(zip([list(range(i)) for i in range(1,5)], ['a','b','c','d'])),
                    columns=['numeros','nombre'])
```

In [58]:
```
df_for_explote
```

Out[58]:

	numeros	nombre
0	[0]	a
1	[0, 1]	b
2	[0, 1, 2]	c
3	[0, 1, 2, 3]	d

In [59]:
```
df_for_explote.explode('numeros')
```

Out[59]:

	numeros	nombre
0	0	a
1	0	b
1	1	b
2	0	c
2	1	c
2	2	c
3	0	d
3	1	d
3	2	d
3	3	d

Merge

El *merge* de pandas es como el *join* de SQL. Nos deja juntar información de dos tablas según la correspondencia de una o más columnas.

```
In [60]: df_personas = pd.DataFrame([[1,'f','maria'],[2,'f','juana'],[3,'m','pedro'],[4,'m','juan'],],
                                    columns = 'id gender name'.split())

In [61]: df_vacaciones = pd.DataFrame([[4,30],[3,20],[7,10]],columns = 'id days'.split())

In [62]: df_personas
```

Out[62]:

	id	gender	name
0	1	f	maria
1	2	f	juana
2	3	m	pedro
3	4	m	juan

```
In [63]: df_vacaciones
```

Out[63]:

	id	days
0	4	30
1	3	20
2	7	10

Merge

```
In [64]: df_personas.merge(df_vacaciones)
```

Out[64]:

	id	gender	name	days
0	3	m	pedro	20
1	4	m	juan	30

```
In [65]: df_personas.merge(df_vacaciones, how='left')
```

Out[65]:

	id	gender	name	days
0	1	f	maria	NaN
1	2	f	juana	NaN
2	3	m	pedro	20.0
3	4	m	juan	30.0

```
In [66]: df_personas.merge(df_vacaciones, how='right')
```

Out[66]:

	id	gender	name	days
0	4	m	juan	30
1	3	m	pedro	20
2	7	NaN	NaN	10

```
In [67]: df_personas.merge(df_vacaciones, how='outer')
```

Out[67]:

	id	gender	name	days
0	1	f	maria	NaN
1	2	f	juana	NaN
2	3	m	pedro	20.0
3	4	m	juan	30.0
4	7	NaN	NaN	10.0

Hay algunos parámetros muy importantes para poder utilizar el merge como queremos. El parámetro *on* nos permite indicar qué columnas tomar para hacer la correspondencia. El parámetro *suffixes* nos sirve para sumar un sufijo a las columnas que se llamen igual.

Join

El *join* es parecido al *merge*, pero trabaja directamente sobre los índices de los DataFrames.

```
In [62]: df_personas
```

Out[62]:

	id	gender	name
0	1	f	maria
1	2	f	juana
2	3	m	pedro
3	4	m	juan

```
In [63]: df_vacaciones
```

Out[63]:

	id	days
0	4	30
1	3	20
2	7	10

```
In [68]: df_personas.join(df_vacaciones, lsuffix='_personas', rsuffix='_vacaciones')
```

Out[68]:

	id_personas	gender	name	id_vacaciones	days
0	1	f	maria	4.0	30.0
1	2	f	juana	3.0	20.0
2	3	m	pedro	7.0	10.0
3	4	m	juan	NaN	NaN

NaNs

Muchas veces vamos a trabajar con datos faltantes. Esto es lo que en Pandas o NumPy se llaman *NaN*. Formalmente significa “Not a Number”, pero también se lo utiliza como dato faltante en columnas de otros tipos.

Las funciones de NumPy y Pandas en general permiten indicar qué comportamiento se quiere tener si aparece un NaN. También hay funciones que directamente trabajan sobre los NaNs tirándolos o reemplazandolos.

```
In [68]: df_con_nans = df_personas.merge(df_vacaciones, how='outer')

In [69]: df_con_nans
```

Out[69]:

	id	gender	name	days
0	1	f	maria	NaN
1	2	f	juana	NaN
2	3	m	pedro	20.0
3	4	m	juan	30.0
4	7	NaN	NaN	10.0

Nans

```
In [70]: df_con_nans.dropna()
```

Out[70]:

	id	gender	name	days
2	3	m	pedro	20.0
3	4	m	juan	30.0

```
In [71]: df_con_nans.dropna(subset=['days'])
```

Out[71]:

	id	gender	name	days
2	3	m	pedro	20.0
3	4	m	juan	30.0
4	7	NaN	NaN	10.0

```
In [72]: df_con_nans.fillna('pepe')
```

Out[72]:

	id	gender	name	days
0	1	f	maria	pepe
1	2	f	juana	pepe
2	3	m	pedro	20.0
3	4	m	juan	30.0
4	7	pepe	pepe	10.0

```
In [73]: df_con_nans['days'] = df_con_nans['days'].fillna(-1)
         df_con_nans
```

Out[73]:

	id	gender	name	days
0	1	f	maria	-1.0
1	2	f	juana	-1.0
2	3	m	pedro	20.0
3	4	m	juan	30.0
4	7	NaN	NaN	10.0

Referencias

Hay miles, algunos buenos:

- https://pandas.pydata.org/pandas-docs/stable/getting_started/tutorials.html
- https://github.com/justmarkham/pandas-videos
- https://www.kaggle.com/learn/pandas
- https://pandas.pydata.org/pandas-docs/stable/reference/index.html

¿Y ahora qué?

Hacer la guia de ejercicios!