Seminario PYTHON - MCAF (Sesión II)

Francisco Gárate Santiago

UCM, 2022

Objetivos de la sesión

Pandas ha llegado a ser la librería de referencia de todo analista de datos, por tanto, su conocimiento es imprescindible.

- Familiarizarse con la librería de estructura de datos (*panel data*) más utilizada.
- Obtener un conocimiento de los usos de esta librería, facilitando su uso frente a otras herramientas como puedan ser el Excel.

Introducción

- **Pandas** (*panel data*) es la extensión de Numpy que ofrece trabajar con estructuras de datos y operar con tablas númericas y series temporales.
- **Sitio**: `https://pandas.pydata.org`
- **Documentación**: `https://pandas.pydata.org/pandas-docs/stable/`
- **Licencia**: open-source BSD 3-Clause License
- El motor de Pandas es **NumPy**, por tanto, las operaciones matriciales son similares e intuitivas.
- En pandas existen dos tipos básicos de objetos, que están basados a su vez en Numpy:
  - Series: vectores 1-D
  - DataFrame: matrices 2-D

Introducción

Operaciones típicas que pueden realizarse fácilmente con **Pandas**:

- Crear un DataFrame a partir de una consulta SQL o SAS, una lista, un archivo externo (csv o xls entre otros), o incluso una página web (url o json).
- Filtrar las filas o columnas de interés
- Limpiar o borrar determinados valores
- Calcular nuevas columnas basada en columnas existentes
- Resumir con una simple función los principales estadísticos de un conjunto de datos
- Graficar una columna contra otra
- Modelar matemáticamente una columna en función de otra, por ejemplo, utilizando regresión lineal.

DataFrame

¿Qué es un **DataFrame**?

- Es una estructura de datos en formato tabla, similar al dataframe disponible en R.
- El formato utilizado en los proyectos de análisis de datos, a pesar de restar velocidad de cálculo (aunque cada nueva versión de Pandas mejora este aspecto).
- Un DataFrame se diferencia de una tabla SQL en que se manipulan directamente en la memoria de su programa ya que no residen en un servidor remoto.
- Son más adecuados para el análisis "offline" complejos con estadísticas, visualización y modelos matemáticos.
- El límite tamaño de un df lo determina la cantidad memoria RAM. Existen ejemplos de implementaciones con 100 millones de filas.

NumPy incluido

Tablas vitaminadas

- Pandas posee como dependencias, entre otras, las librerías **NumPy** y **dateutil**.
- Por tanto, la sintaxis que hemos visto anteriormente en NumPy se aplica exactamente igual en Pandas, pudiendo acceder a las mismas funciones.
- Aun así, es recomendable importar NumPy para operar totalmente con Pandas ya que ambas son perfectamente compatibles.

```
1 import pandas as pd
2 df = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
3 print(df)
4 print(df.mean())
```

```
   0  1  2
0  1  2  3
1  4  5  6
```

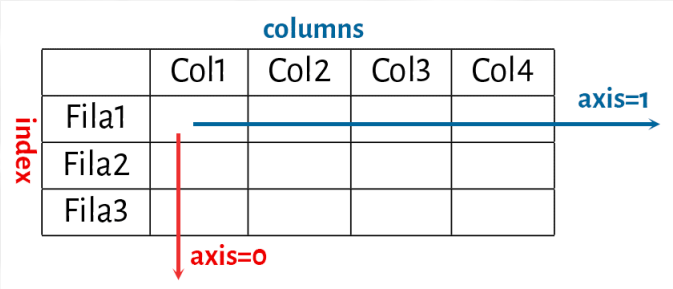
```
0  2.5
1  3.5
2  4.5
```

DataFrame

Axis

Pandas posee las mismas funciones que **Numpy** y sus llamadas son intuitivas. Los DataFrame se componen de filas y columnas. Para cualquier cálculo (por ejemplo la suma), conviene tener en cuenta si las funciones se aplican a la columna *(por defecto)* o a la fila.

- Cada eje (axis) de un DataFrame tiene un índice (index). Por defecto: 0, 1, 2 .. n.
- **axis=0**: Lectura en horizontal. axis=0 puede cambiarse por **axis='index'**
- **axis=1**: Lectura en vertical. axis=1 equivale a **axis='columns'**



DataFrame

Axis

```
   0  1  2
0  1  2  3
1  4  5  6
```

```
1 import pandas as pd  #solo importo pandas
2 df = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
3 print(df.mean())
```

```
0  2.5
1  3.5
2  4.5
```

```
1 print(df.mean(axis=1))  # o axis='columns'
```

```
0  2.0
1  5.0
```

Creación de datos

DataFrame

```
1 df = pd.DataFrame([1, 2, 3, 4], columns=['Importe'])
```

pd.DataFrame(data). Atributos:

- **data**: pueden ser Series, arrays, constants
- **index**: Por defecto será (0, 1, 2, ..., n)
- **columns**: Nombre de las columnas. Por defecto será (0, 1, 2, . . . , n)
- **dtype**: Si no existe, pandas deducirá el tipo. Puede cambiarse posteriormente.

Si el nombre de la columna no contiene espacios, puedo hacer mención a las columnas ya existentes de dos formas:

`df.Importe` *equivale a* `df['Importe']`

Modificación de DataFrame

Creación

Se puede crear un DataFrame vacío o con una sola columna, y posteriormente añadir nuevas.

```
1 import pandas as pd
2 import numpy as np
3 df = pd.DataFrame([1, 2, 3, 4], columns=['Col1'])
4 df['Col2'] = df['Col1'] ** 2
5 print(df)
```

	Col1	Col2
0	1	1
1	2	4
2	3	9
3	4	16

Importación desde texto plano (csv, txt, tab...)

Creación de DataFrame partiendo de un fichero csv.

```
1 import pandas as pd
2 datos = pd.read_csv('fichero.csv', sep=';')
```

pd.read_csv(fichero). Principales parámetros:

- **sep**: Separador en formato *str*. Ejemplo: ';' o '\t' (tabulado). Por defecto: ','
- **decimal**: Separador de decimales. Ejemplo: ',' Por defecto '.'
- **header**: *int*, número de fila donde figura los nombres de las columnas.
- **names**: *str*, Listado con los nombres a usar como cabecera del DataFrame.
- **index_col**: *int*, columna a usar como Index (nombre de las filas) del DataFrame.
- Otros [1]: skiprows, nrows, skipfooter, skip_blank_lines, parse_dates, compression (ficheros zip), memory_map

[1] https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

Importación desde Excel

Creación de DataFrame partiendo de un fichero MS Excel.

```
1 import pandas as pd
2 datos = pd.read_excel('datos.xls')
```

pd.read_excel(fichero). Principales parámetros:

- **sheet_name**: *string*, *int o list*
- **header**: *int*, número de fila con los nombres de las columnas.
- **skiprows**: *int*, indica las primeras filas que deben ignorarse. (Si no hay cabecera)
- **usecols**: *int o list*, indica la columa/s a importar. Ej: 'B:C' o range(1,3)
- **skipfooter**: *int*, indica las filas del final que deben ignorarse. Por defecto: None
- **index_col**: columna a usar como Index del DataFrame.
- **names**: *str*, Listado con los nombres a usar como cabecera del DataFrame.

Importación desde url

Creación de DataFrame partiendo de información publicada en internet. Para ello, Pandas hace uso de las siguientes librerías de *Web scraping*: BeautifulSoup4, htmllib5 o lxml.

pd.read_html(url, flavor). Principales argumentos:

- **url**: dirección url
- **flavor**: bs4, htmllib5 o lxml (por defecto)
- otros: skiprows, header, index_col...

```
1 import pandas as pd
2 web = 'http://acb.com/resultados-clasificacion/ver/'
3 df = pd.read_html(web, index_col='Pos.')
4 print(df[0])
```

Vista y verificación de datos

Una buena costumbre antes de empezar a analizar, es chequear la consistencia de los datos:

- **df.head(n)**: Muestra las n primeras filas del DataFrame/Serie.
- **df.tail(n)**: Muestra las n últimas filas del DataFrame/Serie.
- **df.describe()**: Muestra las principales métricas (conteo, medias, percentiles)
- **df.info()**: Muestra información del df, como el tipo de dato y la memoria utilizada.

Modificar tipos

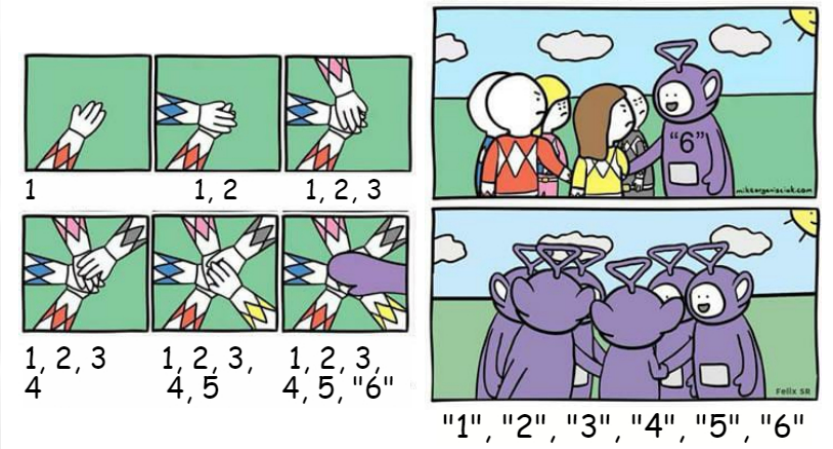
Normalmente, pandas reconoce sin ningún problema el tipo y formato de cada variable que importa. Para cambiar el tipo de variable de cada campo tenemos el comando `.astype()`.

- **.astype(str)**: Convierte a string o texto
- **.astype(int)**: Convierte a entero
- **.astype(float)**: Convierte a número de punto flotante (decimales)
- **.astype('datetime64[]')**: Recomendado para cambiar de uso horario, por ejemplo .astype('datetime64[ns, US/Eastern]'). Para forzar fecha es mejor el uso de pd.to_datetime().

Ejemplo:

```
1 import pandas as pd
2 df = pd.read_csv('Siniestros_2013_2018.csv', sep=';')
3 df['Importe'] = df['Poliza'].astype(int)
```

Modificar tipos



La función lambda combinada con map, reduce o filter sirva hacer conversiones selectivas.

Exportar a CSV o Excel

Pandas, con la misma facilitad que puede leer datos de un fichero csv o Excel, puede exportar cualquier dataframe a dichos formatos con la instrucción **df.to_csv()** o **df.to_excel()**

- Exportación de un df a un fichero csv:

```
1  import pandas as pd
2  import numpy as np
3
4  df = pd.DataFrame(np.zeros(5))
5  df.to_csv('df_empty.csv', sep=';', decimal=',')
```

- Exportación de un df a fichero Excel:

```
1  import pandas as pd
2  web = 'http://acb.com/resultados-clasificacion/ver/'
3  df = pd.read_html(web, index_col='Pos.')
4  df[0].to_excel('acb.xlsx')
```

Limpieza

Data cleaning

Las siguientes instrucciones pueden ser útiles en caso de ser necesario limpiar nuestros datos de valores vacíos, nulos o duplicados:

- **df.fillna(value)**: Rellena los campos NaN con el valor indicado (por ejemplo con 0 o con el valor indicado)

```
df['subp'].fillna(0)
df['sexo'].fillna('M')
df['edad'].fillna(value=df['edad'].mean())
```

- **df.dropna(inplace=True)**: Borra aquellas filas con valores NaN.
  Puede indicarse index=1 para borrar columnas con datos ausentes.
- **df.drop_duplicates(keep='last')**: Borra duplicados, ='first' borra todos excepto el primero *(por defecto)*, ='last' borra todos excepto el último, ='None' borra todos los duplicados.
- **df.index.duplicated()**: Chequea valores index duplicados.

Fechas

Pandas infiere el formato de las columnas importadas. Aun así, a veces es necesario especificar el formato deseado, principalmente cuando manejamos fechas, ya que puede entender un formato erróneo. Así nos aseguramos que Pandas entienda perfectamente la fecha.

Para ello, existe la función **.to_datetime(date, args)** con los siguientes argumentos:

- **date**: en formato *int, float, str, datetime o list*.
  (en algún caso, puede resultar útil forzarlo a ser str con un .astype(str) y aplicarle el formato correcto)
- **format**: Formato del dato origen strftime, ej "`%d/%m/%Y`".
  Los formatos posibles (%d, %m, %Y, %y, %j...) pueden consultarse en:
  `https://docs.python.org/2/library/time.html`
- **errors**: En caso de error, que debe hacer `pandas` ='ignore' (ignorar) o ='coerce' (forzar).

```
1 import pandas as pd
2 df = pd.read_csv('Siniestros_2013_2018.csv', sep=';')
3 df['Fec_ocu'] = pd.to_datetime(df['Fec_ocu'], format='%d-%m-%Y')
```

Fechas

date_range

Una utilidad disponible en Pandas es crear directamente rangos de fechas: **pd.date_range()** con los siguientes parametros:

- **start**: Inicio del rango. Límite izquierdo para generar fechas.
- **end**: Fin del rango. Límite derecho para generar fechas.
- **period**: Número de periodos a generar
- **freq**: Frecuencia de las proyecciones.
  - D: Diaria *(por defecto)*
  - Y: Anual
  - M: Mensual
- **closed**: Si queremos excluir el inicio (closed='right') o el final (closed='left')
- **name**: Nombre del DatetimeIndex resultante *(por defecto ninguno)*

Rango de fechas

date_range

```
1 import pandas as pd
2 s = pd.date_range(start='31/12/2019', periods=8, freq='Y')
3 df = pd.DataFrame(s)
4 print(df)
```

```
            0
0 2019-12-31
1 2020-12-31
2 2021-12-31
3 2022-12-31
4 2023-12-31
5 2024-12-31
6 2025-12-31
7 2026-12-31
```

Rango de fechas

to_period

Por último, una funcionalidad muy útil para agrupar por fechas es convertir/agrupar la fecha en periodos (semanas, meses, trimestres, años). Para ello se utiliza la instrucción **Series.dt.to_period(freq=*str*)**, siendo los principales valores:

- **W**: weekly
- **M**: month end frequency
- **SM**: semi-month end frequency (15th and end of month)
- **SMS**: semi-month start frequency (1st and 15th)
- **Q**: quarter end frequency
- **Y**: year end frequency
- Lista completa: `https://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases`

Rango de fechas

to_period

```
1 import pandas as pd
2 s = pd.date_range('2019-01-01', freq='M', periods=12)
3 p = s.to_period('Q')
4 print(p)
```

```
['2019Q1', '2019Q1', '2019Q1', '2019Q2', '2019Q2', '2019Q2',
 '2019Q3', '2019Q3', '2019Q3', '2019Q4', '2019Q4', '2019Q4']
```

Ejercicio 2

Ejercicio 2

Los datos de la siniestralidad (2013-2018) de una determinada cartera de pólizas se encuentra en el fichero: **Siniestros_2013_2018.csv**. Se pide:

- Importar en un dataframe el fichero .csv y dar formato correcto a las columnas de fecha.
- Convertir las columnas de `Fec_ocu` y `Fec_pago` en frecuencia trimestral ('Q').

Ejercicio resuelto: `https://github.com/franciscogarate/SeminarioPythonUCM`

Añadir columnas

Cuando la columna que queramos añadir sea un cálculo sencillo, o incluso asignarle un valor fijo, simplemente indicamos la variable y la operacion a realizar. Por ejemplo:

```
1 import pandas as pd
2 import numpy as np
3 df = pd.DataFrame([1, 2, 3, 4], columns=['Col1'])
4 df['Col2'] = 100
5 df['Col3'] = np.exp(df['Col1'])
6 print(df)
```

	Col1	Col2	Col3
0	1	100	2.72
1	2	100	4.39
2	3	100	20.09
3	4	100	54.59

Añadir columnas

Otro ejemplo:

```
1 import pandas as pd
2 import numpy as np
3 df = pd.DataFrame([1, 2, 3, 4], columns=['Col1'])
4 df['Col2'] = np.ones(4, dtype=int)
5 df['Col3'] = df['Col1'] + df['Col2']
6 df['Col4'] = np.linspace(1, 10, 4).astype(int)
7 df['Col5'] = 5
8 print(df)
```

	Col1	Col2	Col3	Col4	Col5
0	1	1	2	1	5
1	2	1	3	4	5
2	3	1	4	7	5
3	4	1	5	10	5

Borrar filas o columnas

- Creamos un DataFrame (3x3) con datos aleatorios:

```
1 data = np.random.rand(3, 3)
2 df = pd.DataFrame(data, columns=['Col1','Col2','Col3'])
3 print(df)
```

```
      Col1     Col2     Col3
0 0.738582 0.402065 0.853371
1 0.650983 0.068258 0.738703
2 0.225089 0.598409 0.616579
```

- Borramos una columna:

```
1 df.drop('Col2',axis=1, inplace=True)
2 print(df)
```

```
      Col1     Col3
0 0.738582 0.853371
1 0.650983 0.738703
2 0.225089 0.616579
```

Borrar filas o columnas

- Borramos filas siguiendo una regla para seleccionar el/los index(s) a eliminar:

```
1 df.drop(df[df.Col1 >=0.5].index, inplace=True)
2 print(df)
```

```
       Col1     Col3
2 0.225089 0.616579
```

lambda

Un sola linea para crear una función

- Se conoce a **lambda** como la función anónima (no es exclusiva de pandas ni de python, de hecho acaba de implementarse en MS Excel).

```
1 x = lambda a : a + 10
2 print(x(5))
```

```
15
```

- Una función lambda puede tomar cualquier número de argumentos, pero sólo puede tener una expresión.

```
1 x = lambda a, b, c : a + b + c
2 print(x(3, 4, 5))
```

```
12
```

lambda

- Su uso en pandas facilita la creación de campos calculados:

```
1  import pandas as pd
2  import numpy as np
3
4  df = pd.DataFrame([1, 2, 3, 4], columns=['t'], dtype='int')
5  df['exp'] = df['t'].apply(lambda x: np.exp(x))
6  print(df)
```

	t	exp
0	1	2.718282
1	2	7.389056
2	3	20.085537
3	4	54.598150

lambda

En el caso que sea necesario hacer mención a una función, esta puede llamarse con la instrucción **.apply(lambda x: *funcion*(x))** añadiendo limpieza al código:

```
1  import pandas as pd
2  import numpy as np
3
4  def estacionalidad(x):
5    if x == 7:
6      return 0.9
7    else:
8      return(1.01 ** x)
9
10 df = pd.DataFrame(pd.date_range(start='2020-01-31', periods=12, freq='M',
       name='Fecha'))
11 df['t'] = np.arange(12)
12 df['i'] = 0.02
13 df['factor'] = df['t'].apply(lambda x : estacionalidad(x))
14 print(df)
```

lambda

Como hemos visto, una función lambda puede tomar cualquier número de argumentos, pero sólo puede tener una expresión. Para utilizar dos o más valores del dataframe en la formula, debemos especificar a lambda que lea los valores en horizontal.
Ejemplo: **df.apply(lambda x: f(x.col1, x.col2), axis=1)**

```
1 import pandas as pd
2
3 df = pd.DataFrame([1, 2, 3, 4], columns=['Mes'], dtype="int")
4
5 df['Single Premium'] = df.apply(lambda x: single_risk_premium(x['age'], x
      ['duration']), axis=1)
```

Seleccionar datos

```python
import pandas as pd
df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
        index=[4, 5, 6], columns=['A', 'B', 'C'])
```

	A	B	C
4	0	2	3
5	0	4	1
6	10	20	30

- **df.at[4, 'B']** devuelve `2`
- **df.iat[1, 2]** devuelve `1`
- **df['A']** o **df.A** devuelve `0 0 10`
- **df.loc[5]** devuelve `0 4 1`
- **df.iloc[1]** devuelve `0 4 1`
- **df.loc[5].at['B']** devuelve `4`

Ejercicio 3

Ejercicio 3. Seguro vida entera (flujos anuales)

Crear un dataframe con pd.date_range, inicio 31-12-2020 y final la edad última de las tabla:

- Añadir nueva columna 'edad', siendo 45 años en t=0
- Añadir su lx acorde a la tabla de mortalidad `PASEM2020_Decesos_2ord_Unisex.csv`
- Calcular la probabilidad de fallecimiento (qx).
- Añadir nueva columna 'capital' con 5.000€ constantes.
- Añadir nueva columna 'pagos' calculada como el capital por la probabilidad de fallecimiento.
- Añadir nueva columna con el factor de descuento a utilizar (en este caso, i=0.98%)
- Calcular el NPV de los ingresos descontados con la curva de riesgo

Ejercicio resuelto: `https://github.com/franciscogarate/SeminarioPythonUCM`

Stats

Estadísticos

Como hemos visto, el motor de Pandas estaba basado en NumPy, y por tanto podemos acceder fácilmente a todas las funciones estadísticas (en Series y DataFrames):

- **df.mean()**: media o promedio
- **df.median()**: mediana
- **df.quantile(q)**: Percentil. 0 <= q <=1 (por defecto 0.5)
- **df.max()**: valor máximo
- **df.min()**: valor mínimo
- **df.std()**: desv. estándar. Se puede fijar grados de libertad: ddof=0 o ddof=1 *(por defecto)*
- **df.var()**: varianza

```
1  import pandas as pd
2  import numpy as np
3  s = pd.Series(np.arange(101))
4  df = s.to_frame()
5  print(df.quantile(0.995)) #99.5
```

Stats

describe

- **Series.describe()**: Devuelve los estadísticos básicos.

```
1 import pandas as pd
2 import numpy as np
3 s = pd.Series(np.arange(100))
```

```
count    100.000000
mean      49.500000
std       29.011492
min        0.000000
25%       24.750000
50%       49.500000
75%       74.250000
max       99.000000
```

- Si queremos conocer el valor de un percentil concreto, **Series.quantile(n)** devuelve el n-enesimo percentil siendo n un valor entre 0 y 1. En el ejemplo: **s.quantile(0.995)** devuelve `98.505`

Reindex

En alguna ocasión, es posible que necesitemos reasignar un nuevo *index* a nuestro *dataframe*, o bien, researlo con los valores del 0 ... n. Para ello disponemos de la funcionalidad .reindex():

- **df.reindex()**: resetea los valores por defecto, consecutivos del 0 .. n
- **df.reindex(['A'])**. En este caso, asigna los valores de A como index.

Rename

En otras ocasiones, es posible que necesitemos renombrar los nombres de las columnas. En ese caso podemos utilizar la función **df.rename()**. Los principales argumentos serían:

- mapper: diccionario de nombres o función (ejemplo: str.lower).
- axis=1 (por defecto es 0)
- inplace=True (por defecto es false)

```
1 import pandas as pd
2 df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A','B','C'])
3 df = df.rename({0: 'x', 1: 'y'}, axis=0)
4 df = df.rename({'A': 2, 'B': 4, 'C': 5}, axis=1)
5 print(df)
```

```
   2  4  5
x  1  2  3
y  4  5  6
```

Sampling

Así como obtener muestras aleatorias de los datos de partida con **Series.sample()** con los siguientes atributos:

- **n**: número de muestras
- **frac**: Fraction of axis items to return. ej: 0.1
- **replace**: True (con reemplazamiento) o False (sin reemplazamiento)

```
1 import pandas as pd
2 import numpy as np
3
4 data = pd.Series([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
5 sample = data.sample(3)
6 print(sample)
```

```
1  0.2
4  0.5
2  0.3
```

Bootstrap

- De los n datos originales $y_1, y_2 \dots y_n$, tomar una muestra con reemplazo, de tamaño n.
- Calculamos la media muestral con esa 'pseudomuestra'.
- Repetir B veces. Al final tendremos B estimaciones de la media.

```python
import pandas as pd
import numpy as np

data = pd.Series([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
print(data.describe())

np.random.seed(100)
sample = data.sample(5, replace=True)
statistic = sample.mean()
print(statistic) # 0.2 vs 0.35
```

```
...
mean  0.35
...
```

```
0.2
```

Bootstrap

```
1  import pandas as pd
2  import numpy as np
3
4  data = pd.Series([0.1, 0.2, 0.3, 0.4, 0.5, 0.6])
5  np.random.seed(100)
6
7  b = 1000
8  s = 5
9
10 boot = pd.Series(np.zeros(b))
11 for i in range(b):
12   sample = data.sample(s,replace=True)
13   statistic = sample.mean()
14   boot[i] = statistic
15
16 # Media: 0.348 vs 0.35 + Intervalo confianza 95%:
17 print(boot.mean(), boot.quantile(0.025), boot.quantile(0.975))
```

```
0.34842
```

Agregación de datos: groupby

Para agregar los datos usa **df.groupby(by=['campo']).*funcion*** calculándose, entre otras, las siguientes posibles funciones a la hora de agrupar los datos:

- .sum()
- .count()
- .mean()
- .max()
- .min()
- .std()
- .first()

Se pueden mostrar el estadístico deseado o varios de ellos. En ese caso, la sintaxis debe ser con **.agg()** y **{}** en caso de mostrar varios campos. Por ejemplo:

- df.groupby(['campo']).agg(['min','mean'])
- df.groupby(['campo']).agg({'campo1':'first','campo2':'mean'})

Agregación de datos: groupby

```
1 import pandas as pd
2 df = pd.read_csv("Siniestros_Auto.csv", sep=";", decimal=",")
3 datos = df.groupby(['Delegacion']).agg({'Poliza':'first','Importe':'mean'
      })
4 print(datos)
```

Delegacion	Poliza	Importe
Centro	650372	414.654788
Este	650010	409.912736
Insular	650857	397.174533
Norte	651459	403.703011
Oeste	652037	411.773443
Sur	652726	403.047109

Agregación de datos: groupby

Igualmente podemos agregar varios campos: **df.groupby(['campo1','campo2']).*f()***

En este caso, por defecto muestra las columnas con un formato esquemático o de arból. En caso de querer respetar el formato tabular (necesario para tablas pivotantes) debemos usar **.reset_index()** para que añada un index al inicio.

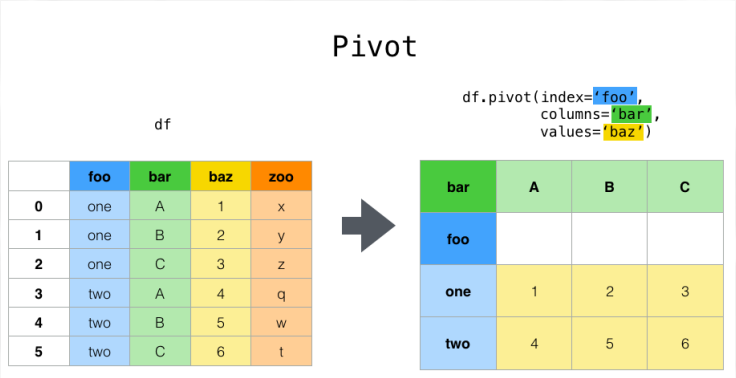
```
1 import pandas as pd
2 df = pd.read_csv("Siniestros_Auto.csv", sep=";", decimal=",")
3 datos = df.groupby(['Delegacion','Categoria']).agg({'Importe':['mean','
      min']}).reset_index()
4 print(datos)
```

```
  Delegacion Categoria Importe
                          mean  min
 0 Centro 1_CAT 418.538364 5.41
 1 Centro 2_CAT 415.717771 2.41
 2 Centro 3_CAT 389.956300 4.57
...
16 Sur    2_CAT 398.954946 8.98
17 Sur    3_CAT 403.195000 5.85
```

Tablas pivotantes

Con Pandas podemos mostrar los datos en dos dimensiones en formato **tabla pivotante** (pivot table). La instrucción **df.pivot()** donde debemos especificar los siguientes campos:

- index: Eje vertical
- columns: Eje horizontal
- values: Valores a mostrar



	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

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

Ejercicio 4

Ejercicio 4

Con los mismos datos que el ejercicio nº 2, se pide:

- Agrupar los datos de siniestralidad por mes de ocurrencia y mes de pago.
- Realizar el triangulo de pagos por periodos mensuales, trimestrales y anuales, utilizando tablas pivotantes.
- Una vez calculado, exportar a un fichero Excel el triangulo anual de pagos acumulado.

Ejercicio resuelto: `https://github.com/franciscogarate/SeminarioPythonUCM`