Apunts CE_5074 7.1

lloc:	Institut d'Ensenyaments a Distància de les Illes Balears	Imprès per:	Carlos Sanchez Recio
Curs:	Sistemes de Big Data	Data:	dimarts, 25 de març 2025, 07:25
Llibre:	Apunts CE_5074 7.1		

Taula de continguts

1. Introducció

En el mòdul de Big Data Aplicat estam veient diverses eines que ens permeten fer feina amb grans conjunts de dades distribuïts en un clúster Hadoop. Hem introduït eines com Pig, amb el seu llenguatge Pig Latin, i Hive, amb el llenguatge HiveQL, que ens permeten fer una anàlisi de les dades emprant una sintaxi semblant a la de SQL.

En els dos lliuraments que ens queden d'aquest mòdul anam a veure com podem escriure programes Python per analitzar dades, és a dir, extreure informació rellevant dels datasets. En aquest lliurament ens centrarem en l'etapa de **preparació i depuració de les dades** (***data wrangling***). Ja hem vist al llarg del curs que és molt habitual que ens trobem amb dades que hem de processar per poder utilitzar-les. O que, fins i tot, contenen dades incorrectes. Veurem aquí algunes tècniques per millorar la qualitat d'aquestes dades, una tasca que haurem de dur a terme sempre abans de fer una anàlisi de dades o d'aplicar un model de *machine learning* com els que s'han vist al mòdul de Sistemes d'Aprenentatge Automàtic. En concret farem feina aquí amb la llibreria *pandas*, que ja vàrem introduir en el mòdul de Programació d'Intel·ligència Artificial. Recordem que les dues estructures de dades principals de *pandas* són les sèries i els dataframes.

Ja serà en el lliurament 8 on farem feina amb les llibreries d'Apache Spark, un *framework* molt popular per a l'anàlisi de dades, que pot funcionar tant dins l'ecosistema Hadoop, com de manera independent. En el lliurament actual del mòdul de Big data aplicat introduirem Spark i la seva arquitectura, mentre que en el lliurament 8 de Sistemes de big data treballarem amb les seves principals llibreries per a l'anàlisi de dades i l'aprenentatge automàtic.

2. Què és data wrangling?

La paraula *wrangling* vol dir una disputa o lluita entre vàries parts, que normalment es perllonga durant un període llarg de temps. Aquest concepte ilustra molt bé el que hem de sufrir amb les dades: una llarga lluita per finalment tenir unes dades que puguin ser emprades per extreure indicadors estadístics rellevants o bé per ser utilitzades en un sistema de *machine learning*.

Així doncs, entenem per ***data wrangling*** totes les tasques que hem de dur a terme per tal de detectar i eliminar errors i inconsistències en les dades originals i organitzar-les, així com combinar dades de diferents fonts, de manera que puguin ser finalment utilitzades en un model estadístic o de *machine learning*. Resumint, definim *data wrangling* com el **procés de neteja, organització i transformació de les dades originals en el format final desitjat per l'analista**.

En català solem traduir *data wrangling* com preparació o depuració o neteja de dades. En anglès també es fan servir els termes *data munging* i *data cleaning*.

La fase de *data wrangling* requereix molt de temps i esforços. Hi ha estudis que indiquen que a prop d'un 80% del temps de l'anàlisi de dades es dedica realment a *data wrangling*. Per tant, essent un procés tan costós, val la pena dedicar-li una mica d'atenció.

Aquí el que veurem són diverses tècniques que se solen aplicar durant el procés de *data wrangling*: tractament dels valors nuls o absents, dels valors repetits, dels valors atípics (*outliers*), normalització (reescalat) de valors o manipulació de cadenes de caràcters (per exemple, per cercar determinats patrons). També veurem com combinar diversos datasets mitjançant concatenacions i *joins*. Per fer tot això treballarem amb la llibreria *pandas* de Python.

A més, tot i que no forma part de la fase de *data wrangling*, en l'apartat 9 veurem com utilitzam diversos mètodes de l'objecte *pandas.DataFrame* per tal d'analitzar un dataset, una vegada ja l'hem netejat.

3. Detecció de valors absents

És molt habitual que en un dataset ens trobem valors buits. En estadística això pot suposar que aquest valor realment no existeix o bé que no s'ha pogut mesurar, per exemple, perquè hi ha hagut algun error. Per exemple, imaginem que tenim un dataset amb la qualitat de l'aire d'una estació de mesurament. Durant uns dies ens trobam que tenim tots els valors de totes les partícules, llevat dels del diòxid de nitrogen. Podria passar que justament en aquests dies no n'hi hagués hagut NO_2 en l'aire, però és poc probable. La causa més probable és un problema en el mesurador.

Per tant, és important diferenciar si un valor en blanc o absent vol dir que era un 0 (no hi havia partícules) o si no es va poder mesurar correctament. Si no tenim això en compte, i prenem els valors absents com un 0, totes les estadístiques ens quedaran desvirtuades: les mitjanes ens sortiran més baixes de la realitat, ens apareixeran períodes de baixa contaminació falsos, etc. A més, ens poden ocasionar problemes amb determinades transformacions de dades, o bé a l'hora d'aplicar un model de *machine learning*. És per això que és important detectar i tractar de manera adequada aquests valors absents.

Hi ha dos enfocaments a l'hora de tractar els valors absents:

1. Podem afegim una llista amb booleans que indiquen si la mesura és correcta o no. Aquest enfocament el podem trobar, per exemple, als datasets de control de la qualitat de l'aire de les Illes Balears. Podem mirar, per exemple, les dades de l'estació del carrer Foners de Palma entre 2016 i 2019 (al portal de dades obertes o al repositori GitHub del curs). Cada columna amb valors de presència de partícules contaminants (per exemple SO2_HI, per al diòxid de sofre), té una altra columna de *flag* que indica si l'anterior és vàlida o no (en aquest cas la columna FL_SO2, on el valor V indica que la dada és vàlida)
2. Podem utilitzar un valor sentinella per especificar explícitament que no tenim dades en aquella posició del dataset. Per exemple, es podria utilitzar -9999 per indicar que no hi ha dades. Així sabem que podem filtrar aquests valors a l'hora de calcular estadístiques.

La llibreria *pandas* segueix aquesta segona aproximació: s'utilitza un valor sentinella, que està definit a la llibreria *numpy*. Es tracta del valor *np.nan* (NaN vol dir *Not a Number*), que es fa servir per representar un valor absent. De fet, el valor NaN està definit a l'especificació dels números amb coma flotant de l'IEEE. Aquest és un valor numèric, però també és freqüent utilitzar-ho per a altres tipus. Per exemple:

```
arbres = pd.Series(['pi', 'alzina', np.nan, 'olivera', 'roure'])
```

Si ara miram quines dades nul·les tenim a la sèrie:

```
arbres.isnull()
```

Tendrem la següent resposta:

```
0 False
1 False
2 True
3 False
4 False
dtype: bool
```

pandas està dissenyat per treballar amb valors absents identificats amb NaN (*np.nan*). Vegem el següent codi en *numpy* per entendre la diferència:

```
import numpy as np
valors = np.array([1, 2, np.nan, 4])
1+ valors[2], valors.sum(), valors.max()
```

El resultat és:

```
(nan, nan, nan)
```

perquè *nunpy* no està preparat per fer feina amb valors absents i, per tant, quan feim qualsevol operació amb un NaN, el resultat sempre és NaN.

En canvi, vegem el següent codi fent servir *pandas*:

```
import pandas as pd
serie = pd.Series([1, 2, np.nan, 4])
serie.sum(), serie.max()
```

El resultat és:

```
(7.0, 4.0)
```

perquè *pandas* sap que no ha de tenir en compte els valors absents identificats amb *np.nan* quan calcula la mitjana o el màxim: només es fan servir els valors no nuls.

En qualsevol cas, com comentàvem, quan treballam amb un model d'aprenentatge automàtic, potser sí que ens poden aparèixer problemes amb els valors absents, i per això és important tractar prèviament aquests valors de forma adequada. En aquests casos, hem de detectar els valors absents i substituir-los per un altre valor. Segons el context, per evitar distorsions, pot ser útil eliminar tots els valors NaN de les dades. O reemplaçar cada valor NaN pel seu predecessor o successor en la sèrie o dataframe. Un altre enfocament habitual és substituir-los per la mitjana de tots els valors no nuls. En lloc de la mitjana pot utilitzar-se un altre indicador de centralitat dels que vàrem veure en el lliurament 2, com ara la mediana o la moda.

Per fer aquestes operacions, tenim els següents mètodes en la llibreria *pandas*:

- *isnull*(): retorna un objecte amb booleans de la mateixa mida que la sèrie o dataframe que indica si cada posició és un valor absent (NaN) o no. És a dir, genera una màscara booleana que indica els valors absents (amb *true*). N'hem vist un exemple abans.
- *notnull*(): és el contrari que *isnull*()
- *dropna*(): retorna una versió filtrada de les dades, eliminant els valors absents
- *fillna*(): retorna una còpia de les dades on es substitueixen els valors absents per un altre valor concret

Supressió dels valors absents

Vegem un exemple per esborrar els valors absents de la sèrie que hem definit anteriorment:

```
serie_neta = serie.dropna()
```

En canvi, quan treballam amb dataframes, no podem esborrar un valor aïllat, ho hem de fer per tota la seva fila (per defecte) o columna. Vegem un exemple:

```
dades = pd.DataFrame([[1, 2, 3], [4, np.nan, 6], [7, 8, 9]])
dades_netes = dades.dropna()
dades_netes
```

El resultat és que s'elimina tota la segona fila:

	0	1	2
0	1	2.0	3
2	7	8.0	9

Si en canvi volguéssim eliminar la columna, hem de passar axis=1 com a argument:

```
dades_netes = dades.dropna(axis=1)
```

També podem especificar que només s'esborri una fila o columna només en el cas en què tots els seus valors siguin NaN. Per a files, seria:

```
dades_netes = dades.dropna(how='all')
```

I per a columnes:

```
dades_netes = dades.dropna(axis=1,how='all')
```

Substitució dels valors absents

Tal i com hem comentat, una altra manera de tractar aquests valors absents és substituir-los per un valor no nul, emprant el mètode *fillna*. Per exemple, podem substituir-lo per la mitjana dels valors no nuls:

```
dades2 = dades.fillna(dades.mean())
```

El resultat és que la cel·la [1,1] es substitueix pel valor 5.0:

	0	1	2
0	1	2.0	3
1	4	5.0	6
2	7	8.0	9

És habitual també substituir el valor absent pel seu precedent o pel seu successor. Per fer-ho, especificam el mètode *ffill* (*forward-fill*) si volem propagar el valor cap endavant (és a dir, emprar el predecessor) o *bfill* (*back-fill*) per propagar el valor cap enrere (és a dir, emprar el successor). Per exemple:

```
dades2 = dades.fillna(method='ffill')
```

substitueix el valor absent (posició [1,1]) pel valor de la mateixa columna en la fila anterior, és a dir un 2:

	0	1	2
0	1.0	2.0	3.0
1	4.0	4.0	6.0
2	7.0	8.0	9.0

Podríem especificar que ho fes per la mateixa fila, en la columna anterior, amb axis=1, és a dir, el substituiria per 4:

```
dades2 = dades.fillna(method='ffill',axis=1)
```

Hi ha vegades on ens interessa substituir els valors absents per valors diferents depenent de la columna. Pensem en un dataframe amb dades de persones on tenim una columna per a l'alçada i una altra per al pes: no té sentit substituir els valors absents pel mateix valor per a l'altura i per al pes. Podríem fer-ho per la mitjana de cada columna:

```
dades = [[185, 75], [160, 55], [177, np.nan], [np.nan, 68]]
df = pd.DataFrame(dades, columns = ['altura', 'pes'])
df2 = df.fillna({'altura': df['altura'].mean(), 'pes': df['pes'].mean()})
df2
```

Aquest és el resultat:

	altura	pes
0	185.0	75.0
1	160.0	55.0
2	177.0	66.0
3	174.0	68.0

Per últim, si el dataset utilitza un altre valor sentinella, per exemple el -9999, podem emprar el mètode *replace*. Per exemple, amb aquesta línia substituïm el valor -9999 per un NaN dins la sèrie o dataframe *dades*:

```
dades.replace(-9999, np.nan)
```

```

```

4. Detecció d'outliers

Recordem que en el lliurament 2 vàrem introduir el concepte d'*oulier* o valor atípic. Recordem també que vàrem veure com un *outlier* feia que els estudis amb major nivell d'ingressos de la Universitat de Carolina del Nord fos Geografia, perquè eren els d'en Michael Jordan.

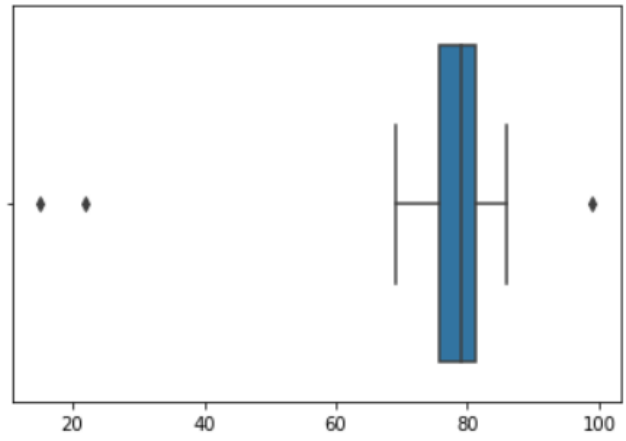
En alguns dels datasets que hem vist durant el curs també hem pogut apreciar la presència d'*outliers*. Per exemple, en els allotjaments d'Airbnb a Menorca vàrem veure com un petit grup d'allotjaments amb un preu molt alt feia que el preu mitjà pujàs per damun dels 320 euros per nit.

Vegem un exemple encara més clar. Suposem que a un dels municipis de Menorca tenim 100 allotjaments. 99 d'ells tenen un preu de 100 euros la nit, però n'hi ha un amb un preu de 100.000 euros, ja sigui per error o perquè el propietari volia fer la gràcia que la seva és la casa més cara de l'illa. Tot i que només hi ha un valor diferent de 100 euros, la mitjana és de 1.099 euros la nit! Si no feim res amb aquest valor, qualsevol indicador estadístic (per exemple, el municipi més car) sortirà completament desvirtuat. I si volem emprar aquestes dades per a un model de *machine learning*, aquest *outlier* també tendrà un gran efecte distorsionador.

Una manera senzilla de detectar els *outliers* és visualment. Ja hem vist alguns exemples amb l'histograma o amb el diagrama de punts, quan treballam amb dues variables. Un altre diagrama que és de gran ajuda en aquests casos és el de caixa o *boxplot* (que ja coneixem del lliurament 5). Vegem-ne un exemple, emprant la llibreria *seaborn*.

```
import seaborn as sns
dades = np.array([78,81,99,77,69,83,82,71,15,80,79,77,76,22,75,85,86,79,80,81])
sns.boxplot(x=dades)
```

El resultat és aquest gràfic:



Podem veure clarament que hi ha tres *outliers* en aquestes dades, 2 per sota (15 i 22) i un per dalt (99). Una vegada detectats, podem analitzar-los i, normalment, acabarem eliminant-los o bé modificant-los.

Detecció automàtica

Tot i que els gràfics com el *boxplot* que acabam de veure són molt útils, requereixen la intervenció d'una persona per interpretar-los. Molt sovint, especialment si treballam amb grans conjunts de dades, voldrem automatitzar aquest procés de detecció dels *outliers*. En aquests casos emprarem la mitjana i la desviació estàndard: considerarem un outlier a aquell valor que estigui separat de la mitjana en més de *x* vegades la desviació estàndard. Aquesta *x* la podem triar nosaltres, en funció de si volem ser més o menys conservadors: una *x* petita (per exemple 1 o 2) detectarà molts *outliers*, mentre que una *x* més gran (per exemple 5), en detectarà pocs; un valor bastant habitual és 3.

Vegem un exemple amb la llista *dades* de l'exemple anterior. Com que aquí la llista és molt petita, la desviació estàndard és més gran de l'habitual. Així que triarem *x=1*, és a dir, tot el que estigui a més d'una vegada la desviació estàndard de la mitjana es considerarà un *outlier*.

```
df = pd.Series(dades)
df[np.abs(dades-df.mean()) > (df.std())]
```

El resultat és que detecta els 3 *outliers* que cercàvem (99, 15 i 22):

```
2     99
8     15
13    22
dtype: int32
```

```

```

Modificació o supressió dels outliers

Una vegada hem identificat els outliers, hem de tractar-los. Igual que amb els valors absents tenim dues opcions: eliminar-los o substituir-los per un valor representatiu. Això no vol dir que modifiquem les dades originals, on ens pot interessar continuar guardant aquests valors atípics. Però no volem utilitzar-los per als càlculs estadístics ni per als models de *machine learning*.

El més habitual és substituir els valors atípics per la mitjana més o menys *x* vegades la desviació estàndard, amb la *x* que hàgim emprat per detectar-los.

5. Detecció de duplicats

També és freqüent trobar files duplicades dins d'un dataframe. Aquestes files de vegades provenen d'errors. En qualsevol cas, encara que no sigui així, molt sovint no les volem tenir en compte a l'hora d'aplicar un algorisme de *machine learning*, com per exemple un *clustering*.

L'objecte *DataFrame* conté un mètode *duplicated* que retorna una sèrie de booleans que indica si una fila és un duplicat o no. Vegem-ne un exemple:

```
dades = pd.DataFrame({'columna1': ['a', 'b', 'c', 'a', 'a', 'b', 'a', 'c'],
                      'columna2': [1, 3, 3, 2, 1, 2, 3, 3]})
dades.duplicated()
```

I el resultat és:

```
0    False
1    False
2    False
3    False
4     True
5    False
6    False
7     True
dtype: bool
```

L'objecte *Series* també té el mètode *duplicated* per detectar els valors duplicats.

Supressió de duplicats

Una vegada hem detectat la presència de duplicats, és molt probable que els vulguem eliminar. Per això podem emprar el mètode *drop_duplicates* dels objectes *Series* i *DataFrame*. Vegem-ne un exemple:

```
dades2 = dades.drop_duplicates()
dades2
```

I el resultat és que s'ha eliminat la fila 7 perquè estava duplicada:

	columna1	columna2
0	a	1
1	b	3
2	c	3
3	a	2
5	b	2
6	a	3

Podem triar quina volem mantenir d'entre totes les duplicades, emprant el paràmetre *keep*, que pot prendre el valor 'first' (la primera que es trobi), 'last' (la darrera) o False (no se'n manté cap, s'esborren totes). El valor per defecte és 'first', tal i com hem vist en l'exemple anterior.

En dataframes grans amb moltes columnes, quan volem aplicar un algorisme de *machine learning*, és freqüent que només ens interessin dues (o unes poques) columnes. El mètode *drop_duplicates* permet eliminar les files que tenguin valors duplicats només en aquestes columnes. Per exemple, si un dataframe té les columnes c1, c2, c3, c4, c5, c6, c7 i c8, suposem que volem eliminar les files on les columnes c2 i c6 (les que ens interessen per al nostre model de machine learning) estan duplicades. Ho escriuríem així:

```
df.drop_duplicates(['c2', 'c6'])
```

6. Reescalat de les dades

Suposem que tenim un dataframe amb dades d'alçada i pes d'una població. Volem aplicar un algorisme de *clustering* per detectar grups. Les unitats en què estiguin definides les dades influiran en el resultat del *clustering*. Les dades d'alçades poden estar en centímetres o en polzades, mentre que les de pes en kilograms o lliures. I depenent de quina combinació emprem, és molt probable que ens surtin diferents clústers.

Vegem un exemple:

Persona	Alçada (cm)	Alçada (polzades)	Pes (Kg)	Pes (lliures)
A	160	63	68	150
B	170	66,9	72	158,7
C	176	69,3	77	169,8

Taula: *Alçades i pesos d'una població de tres individus*

Anam a veure quines són les distàncies entre ells, emprant una distància euclídea, si agafam les alçades en cm i els pesos en Kg:

```
from scipy.spatial import distance
a_b = distance.euclidean([160, 68], [170, 72])
a_c = distance.euclidean([160, 68], [176, 77])
b_c = distance.euclidean([170, 72], [176, 77])
a_b, a_c, b_c
```

El resultat és:
(10.770329614269007, 18.35755975068582, 7.810249675906654)

Per tant, el veïnat més proper a l'individu B és C (amb una distància de 7.810249675906654)

Ara anam a veure els resultats si ho feim amb polzades i lliures:

```
a_b = distance.euclidean([63, 150], [66.9, 158.7])
a_c = distance.euclidean([63, 150], [69.3, 169.8])
b_c = distance.euclidean([66.9, 158.7], [69.3, 169.8])
a_b, a_c, b_c
```

El resultat és:

(9.534149149242413, 20.77811348510736, 11.35649593844864)

Podem veure que el veïnat més proper a B ara no és C, sinó que és A (amb una distància de 9.534149149242413).

Aleshores, si els veïnats més propers són diferents depenent de les unitats emprades, ja podem veure clarament que els *clústers* resultants poden també ser diferents.

En cassos com aquests, on les dimensions (les columnes) utilitzen unitats de mesura diferents, és convenient fer un procés de reescalat de les dades. Per fer-ho, anam a normalitzar les dues columnes, de manera que ambdues tenguin una mitjana de 0 i una desviació estàndard de 1. Per fer-ho, a cada valor li restarem la mitjana i el dividirem per la desviació típica:

$$z = \frac{x - \mu}{\sigma}$$

on x és un valor de la llista, μ és la mitjana de la llista, σ la desviació estàndard i z és el corresponent valor normalitzat. Si us fixau bé, aquesta és la mateixa fórmula que vàrem emprar per normalitzar una distribució normal en el lliurament 2.

D'aquesta manera, llevam les unitats i convertim cada dimensió en "número de desviacions estàndard respecte de la mitjana".

Vegem com implementar una funció per reescalar d'aquesta manera una llista:

```
import numpy as np
from typing import List
def reescalar(dades) -> List[float]:
    m = np.mean(dades)
    s = np.std(dades)
    m, s
    reescalat = dades.copy()
    if s != 0:
        for i in range(0,len(reescalat)):
            reescalat[i] = (reescalat[i]-m)/s
    return reescalat
```

Si ara reescalam la nostra llista d'alçades en cm:

```
noves_altures = reescalar([160,170,176])
noves_altures
```

Obtenim la següent llista:

```
[-1.3131983079178726, 0.2020305089104436, 1.1111677990074333]
```

Tal i com hem dit abans, aquesta nova llista no té unitats i ens indica quantes vegades la desviació estàndard està cada valor separat de la mitjana. De fet, si no hi haguessin petites desviacions per l'arrodoniment (no hem emprat tots els decimals en les conversions entre cm i polzades), els valors haurien de ser els mateixos si aplicàssim la normalització per a les alçades en polzades.

Aquesta llista normalitzada (i la corresponent als pesos) seria la que hauríem d'emprar per a l'algorisme de *clustering*.

7. Manipulació de strings

Fins ara hem xerrat de manipular valors numèrics, que és el més habitual quan volem fer càlculs estadístics o aplicar un algorisme de *machine learning*. Però de vegades també és necessari tractar cadenes de caràcters.

El llenguatge Python incorpora moltes operacions per al processament de strings. A més, per a transformacions més complexes, podem emprar expressions regulars. *pandas* afegeix la possibilitat d'aplicar operacions amb strings i expressions regulars sobre el conjunt d'una sèrie o dataframe (a més de donar suport a la detecció de valors absents).

Fragmentar un string

Una de les operacions més habituals amb strings és la de particionar una cadena en funció d'un caràcter separador, mitjançant el mètode *split*. Per exemple:

```
s = 'a,   b ,     cd'
s.split(',')
```

El resultat és un array de 3 posicions:

```
['a', '   b ', '     cd']
```

El mètode *split* sol utilitzar-se juntament amb *strip* per tal de llevar els caràcters en blanc (i salts de línia) que hi ha al començament o final de cada cadena:

```
s = 'a,   b ,     cd'
s2 = [x.strip() for x in s.split(',')]
s2
```

El resultat és: `['a', 'b', 'cd']`

Suposem que ara volem obtenir un nou string que empri el caràcter dos punts per separar els diferents valors. Ho podem fer amb el mètode *join*:

```
s3 = ':'.join(s2)
```

Dóna la cadena `'a:b:cd'`.

Cercar valors en un string

La manera més senzilla és utilitzar l'operador *in*, tot i que també podem emprar els mètodes *find* i *index*:

```
'cd' in s3
```

retorna True.

```
s3.find('cd')
```

retorna 4, la posició de *s3* on troba la cadena 'cd'. Si no troba la cadena 'cd' retorna -1.

```
s3.index('cd')
```

També retorna 4, però amb una diferència: si no troba la cadena 'cd' produeix una excepció.

Per últim, també podem comptar quantes vegades apareix una cadena dins d'una altra, mitjançant el mètode *count*:

```
s3.count('b')
```

retorna 1.

Substituir valors en un string

Tal i com ja hem vist anteriorment, tenim el mètode replace per reemplaçar totes les ocurrències d'una subcadena dins d'una altra. Per exemple:

```
s = 'a,b,cd'
s2 = s.replace(',',':')
s2
```

Ens dona la cadena `'a:b:cd'`

Altres mètodes dels strings

A la documentació de Python, en l'apartat de tipus *built-in*, podeu trobar els detalls dels mètodes del tipus string: https://docs.python.org/3/library/stdtypes.html#string-methods

Expressions regulars

De vegades volem fer transformacions amb cadenes de caràcters més complexes. Per fer-ho, podem emprar les expressions regulars. El mòdul *re* és el responsable d'aplicar expressions regulars sobre strings en Python. A https://docs.python.org/3/library/re.html podeu trobar la documentació detallada sobre el mòdul *re*. Ara anam a mostrar un petit exemple, on treballam amb una expressió regular per a la validació de les adreces de correu: ([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4}). Vegem el següent codi:

```
text = """Joan joan@gmail.com
Aina aina@hotmail.com
Pep pep@yahoo
Maria maria@mail.xyz.com"""
patro = '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'
regex = re.compile(patro, flags=re.IGNORECASE)
regex.search(text)
```

Podem veure que primer assignam la cadena amb l'expressió regular a una variable *patro*. Després, amb el mètode *compile* de *re* generam l'objecte *regex* per gestionar l'expressió regular, sobre el qual podrem utilitzar diversos mètodes per cercar. Hem emprat el flag *IGNORECASE* perquè no faci distinció entre majúscules i minúscules. Per últim, feim servir el mètode *search* per trobar la primera coincidència del patró que cercam. Aquesta coincidència es representa mitjançant un objecte de tipus *re.Match*. El resultat d'aquest codi és:

```
<re.Match object; span=(5, 19), match='joan@gmail.com'>
```

A més de *search*, altres mètodes són útils per a cercar patrons en un text. *findall* ens retorna una llista amb totes les coincidències (no superposades) que satisfan el patró. A més, cada coincidència es representa mitjançant una tupla que conté els diversos grups que hem especificat en el patró. Seguint amb el nostre exemple:

```
regex.findall(text)
```

retorna:

```
[('joan', 'gmail', 'com'),
 ('aina', 'hotmail', 'com'),
 ('maria', 'mail.xyz', 'com')]
```

El mètode *finditer* és similar a *findall*, però retorna un iterador en lloc d'una llista. Per últim, el mètode *match* ens indica si troba una coincidència al principi del text. Si la troba, ens retorna un objecte de tipus *re.Match* amb la coincidència. En cas contrari, ens retorna *None*. Seguint amb el nostre exemple:

```
regex.match(text)
```

Ens retorna **None**. En canvi,

```
print(regex.match("joan@gmail.com"))
```

Ens retorna:

```
<re.Match object; span=(0, 14), match='joan@gmail.com'>
```

Operacions amb strings vectoritzats

Netejar un dataset sovint requereix manipular strings. Però, com ja sabem, les columnes amb strings també és freqüent que contenguin dades absents.

Anam a treballar ara amb aquest objecte Series:

```
dades = pd.Series({'Joan': 'joan@gmail.com', 'Aina': 'aina@hotmail.com',
 'Pep': 'pep@yahoo', 'Maria': np.nan})
```

Podem cercar els valors absents mitjançant *isnull*, igual que hem vist abans:

```
dades.isnull()
```

retorna:

```
Joan     False
Aina     False
Pep      False
Maria     True
dtype: bool
```

Podríem emprar *dades.map* per cercar subcadenes (o expressions regulars) en cada un dels valors de la sèrie. Però fallaria quan es trobin un valor absent *np.nan*. Per evitar això, l'objecte *Series* té un atribut *str* que ens permet treballar amb els strings que conté. Tots aquests mètodes estan preparats per treballar amb els valors *np.nan*. Per exemple, si volem cercar els elements que contenen la cadena *hotmail*, ho faríem així:

```
dades.str.contains('hotmail')
```

que ens retorna:

```
Joan     False
Aina      True
Pep      False
Maria      NaN
dtype: object
```

I també podem fer-ho servir per treballar amb expressions regulars. Per exemple, amb *findall* cerca totes les coincidències del patró que hem vist abans:

```
dades.str.findall(patro, flags=re.IGNORECASE)
```

retorna:

```
Joan       [(joan, gmail, com)]
Aina     [(aina, hotmail, com)]
Pep                         []
Maria                      NaN
dtype: object
```

D'una manera semblant, també podem emprar el mètode *match*, que ens diu per a cada posició si es troba o no el valor cercat:

```
dades.str.match(patro, flags=re.IGNORECASE)
```

que retorna:

```
Joan      True
Aina      True
Pep      False
Maria      NaN
dtype: object
```

Per acabar amb aquest capítol, vegem el cas més habitual, on tenim un dataframe amb diverses columnes. Per exemple:

```
persones = [('Joan', 'joan@gmail.com'), ('Aina', 'aina@hotmail.com'),
 ('Pep', 'pep@gmail'), ('Maria', np.nan)]
dades = pd.DataFrame(data=persones, columns=['nom', 'correu'])
```

Ara podem utilitzar el que hem vist sobre l'atribut *str* d'una sèrie (o columna) per cercar el nostre patró:

```
dades['correu'].str.match(patro, flags=re.IGNORECASE)
```

que ens retorna:

```
0     True
1     True
2    False
3      NaN
Name: correu, dtype: object
```

8. Combinar datasets

Moltes vegades, per dur a terme una bona anàlisi necessitam combinar dades de diferents fonts. Aquestes operacions de combinar dos (o més) datasets diferents poden ser de dos tipus:

- concatenar els dos datasets, les files d'un dataset darrera de les de l'altre
- fusionar dos datasets fent servir un *join* semblant al de les bases de dades relacionals

Els objectes *Series* i *DataFrames* estan dissenyats tenint en compte aquestes operacions, de manera que *pandas* incorpora mètodes per dur-les a terme d'una manera senzilla i eficient: són el mètode *concat* per implementar la concatenació i el mètode *merge* per implementar els *joins*. A continuació veurem els detalls.

8.1. Concatenacions

La concatenació de sèries i dataframesfuncionen d'una manera semblant a la d'arrays de *numpy*. En el cas d'arrays d'una única dimensió:

```
import numpy as np
a = [1, 2, 3]
b = [4, 5, 6]
c = [7, 8, 9]
np.concatenate([a, b, c])
```

El resultat és un array amb els tres elements de l'array *a*, seguits dels 3 elements de l'array *b* i dels 3 elements de l'array c:

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Si tenim dues dimensions, funciona d'una d'una manera molt semblant:

```
a = [[1, 2], [3, 4]]
b = [[5, 6], [7, 8]]
np.concatenate([a, b])
```

El resultat és les dues files de l'array *a* seguides de les dues files de l'array *b*:

```
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

Podem també fer la concatenació per columnes, especificant-ho amb el paràmetre *axis=1*. D'aquesta forma, la primera fila de *b*, s'afegeix al final de la primera fila d'*a*, i el mateix per a la segona fila:

```
np.concatenate([a, b], axis=1)
```

El resultat és:

```
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

Una vegada feta aquesta introducció, anam a veure com funciona amb *Series*, on emprarem la funció *pd.concat*:

```
import pandas as pd
a = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
b = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([a,b])
```

El resultat és una concatenació, primer els elements de la sèrie *a* i després els de la *b*:

```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

La principal diferència amb *np.concatenate* és que amb *pd.concat* es respecten els índexs. Això ens pot donar problemes ja que pot ocasionar tenir índexs duplicats dins de la sèrie resultant. Vegem-ne un exemple on no hem especificat els índexs (i per tant van de 0 a 2 en les dues sèries):

```
a = pd.Series(['A', 'B', 'C'])
b = pd.Series(['D', 'E', 'F'])
pd.concat([a,b])
```

Podem veure que en el resultat tenim dos índex 0, dos índexs 1 i dos índexs 2:

```
0    A
1    B
2    C
0    D
1    E
2    F
dtype: object
```

Per corregir aquesta duplicació d'índexs tenim tres opcions. La primera és afegir el flag *verify_integrity=True* de manera que es llenci una excepció si apareix qualque índex repetit:

```
try:
    pd.concat([a, b], verify_integrity=True)
except ValueError as e:
    print("Error d'integritat:", e)
```

El resultat serà que es llançarà la següent excepció:

```
Error d'integritat: Indexes have overlapping values: Int64Index([0, 1, 2],
 dtype='int64')
```

La segona opció és especificar que s'ignorin els índexs quan es faci la concatenació, amb el flag *ignore_index=True*:

```
pd.concat([a, b], ignore_index=True)
```

Podem veure que ja no tenim índexs duplicats:

```
0    A
1    B
2    C
3    D
4    E
5    F
dtype: object
```

La tercera opció consisteix en definir un índex multinivell. En el següent exemple, afegirem un nivell d'índex amb el valor 'a' per al primer dataset i amb el valor 'b' per al segon:

```
pd.concat([a,b], keys=['a', 'b'])
```

Podem veure que com que ara tenim dos nivells d'índex, ja no hi ha valors repetits:

```
a  0    A
   1    B
   2    C
b  0    D
   1    E
   2    F
dtype: object
```

Podem també concatenar objectes *DataFrame* amb la funció *pd.concat*:

```
a = pd.DataFrame([['A1', 'A2'], ['B1','B2']], index=[1, 2])
b = pd.DataFrame([['A3', 'A4'], ['B3','B4']], index=[3, 4])
pd.concat([a,b])
```

Com podem veure, per defecte la concatenació es fa per files, i els índexs també es conserven:

	0	1
1	A1	A2
2	B1	B2
3	A3	A4
4	B3	B4

Igual que amb *np.concatenate*, també podem fer la concatenació per columnes. Vegem-ne un exemple:

```
a = pd.DataFrame([['A1', 'A2'], ['B1','B2']])
b = pd.DataFrame([['A3', 'A4'], ['B3','B4']])
pd.concat([a,b], axis=1)
```

On el resultat és (notau que tenim índexs repetits en les columnes):

	0	1	0	1
0	A1	A2	A3	A4
1	B1	B2	B3	B4

Notau que tenim índexs repetits en les columnes. Podríem solucionar-ho amb qualsevol de les tres opcions que abans hem vist. Per exemple, amb el flat ignore_index:

```
pd.concat([a,b], axis=1, ignore_index=True)
```

que dóna com a resultat:

	0	1	2	3
0	A1	A2	A3	A4
1	B1	B2	B3	B4

8.2. Joins

La funció *pd.merge* és la interface que permet dur a terme els joins típics de l'àlgebra relacional. Precisament, una de les principals característiques de *pandas* és que implementa aquests joins entre dos dataframes en memòria i d'una forma eficient. Tal i com veurem en els següents exemples, podem treballar amb diferents tipus de joins segons la seva cardinalitat: un-a-un, molts-a-un i molts-a-molts.

Començam amb els dos dataframes següents:

```
df1 = pd.DataFrame({'empleat': ['Joan', 'Aina', 'Maria', 'Pep'],
'departament': ['Desenvolupament', 'Sistemes', 'Administració', 'Desenvolupament']})
df2 = pd.DataFrame({'empleat': ['Maria', 'Aina', 'Pep', 'Joan'],
'anycontractacio': [2021, 2022, 2021, 2020]})
```

df1	empleat	departament
0	Joan	Desenvolupament
1	Aina	Sistemes
2	Maria	Administració
3	Pep	Desenvolupament

df2	empleat	anycontractacio
0	Maria	2021
1	Aina	2022
2	Pep	2021
3	Joan	202

I ara feim un join:

```
df3 = pd.merge(df1, df2)
```

Aquí la funció *pd.merge* reconeix que tots dos dataframes tenen una columna "empleat" i automàticament fa el join amb aquesta columna com a clau. El resultat és un nou dataframe que combina els dos anteriors:

df3	empleat	departament	anycontractacio
0	Joan	Desenvolupament	2020
1	Aina	Sistemes	2022
2	Maria	Administració	2021
3	Pep	Desenvolupament	2021

Podria passar que tenguéssim una altra columna als dos dataframes amb el mateix nom. Per evitar problemes, és una bona pràctica utilitzar la clàusula ***on*** per especificar sobre quina columna volem fer el join:

```
df3 = pd.merge(df1, df2, on='empleat')
```

No sempre passarà com en aquest exemple que les dues columnes que volem emprar per fer el join tenen el mateix nom. Si no és així, hem d'especificar-les mitjançant ***left_on*** i ***right_on***, com en el següent exemple:

```
df1 = pd.DataFrame({'empleat': ['Joan', 'Aina', 'Maria', 'Pep'],
'departament': ['Desenvolupament', 'Sistemes', 'Administració', 'Desenvolupament']})
df2 = pd.DataFrame({'treballador': ['Maria', 'Aina', 'Pep', 'Joan'],
'anycontractacio': [2021, 2022, 2021, 2020]})
df3 = pd.merge(df1, df2, left_on='empleat', right_on='treballador')
```

El resultat és aquest dataframe:

	empleat	departament	treballador	anycontractacio
0	Joan	Desenvolupament	Joan	2020
1	Aina	Sistemes	Aina	2022
2	Maria	Administració	Maria	2021
3	Pep	Desenvolupament	Pep	2021

Podem veure que les columnes empleat i treballador són iguals, així que en podríem esborrar una de les dues:

```
df3.drop('treballador', axis=1, inplace=True)
```

	empleat	departament	anycontractacio
0	Joan	Desenvolupament	2020
1	Aina	Sistemes	2022
2	Maria	Administració	2021
3	Pep	Desenvolupament	2021

En aquests exemples, la columna que s'ha emprat com a clau per al join en els dos dataframes només té valors únics, no repetits: és un join de tipus **un-a-un**. Però això no sempre serà així. En el següent exemple anam a emprar la columna '*departament*' del dataframe *df1*, que sí que té valors duplicats, per fer el join. Així doncs, tendrem un join de tipus **molts-a-un**.

```
df4 = pd.DataFrame({'departament': ['Desenvolupament', 'Administració', 'Sistemes'],
'cap': ['Jaume','Mar','Rosa']})
df5 = pd.merge(df1, df4)
```

El resultat és:

	empleat	departament	cap
0	Joan	Desenvolupament	Jaume
1	Pep	Desenvolupament	Jaume
2	Aina	Sistemes	Rosa
3	Maria	Administració	Mar

Per últim, vegem un exemple de tipus molts-a-molts, on en els dos costats del join podem trobar valors duplicats. Anam a emprar un nou dataframe que té els requeriments de feina que han de tenir els treballadors de cada departament:

```
df6 = pd.DataFrame({'departament': ['Desenvolupament', 'Desenvolupament',
 'Administració', 'Sistemes', 'Sistemes'],
'requeriment': ['Java', 'Python',
 'Fulls de càlcul', 'Linux', 'Windows Server']})
```

	departament	requeriment
0	Desenvolupament	Java
1	Desenvolupament	Python
2	Administració	Fulls de càlcul
3	Sistemes	Linux
4	Sistemes	Windows Server

El resultat del join per departament és:

```
df7 = pd.merge(df1, df6)
```

	empleat	departament	requeriment
0	Joan	Desenvolupament	Java
1	Joan	Desenvolupament	Python

	empleat	departament	requeriment
2	Pep	Desenvolupament	Java
3	Pep	Desenvolupament	Python
4	Aina	Sistemes	Linux
5	Aina	Sistemes	Windows Server
6	Maria	Administració	Fulls de càlcul

Tipus de joins

Hem vist que, depenent de si tenim o no valors duplicats en les columnes que intervenen en el join, tenim joins de tipus un-a-un, molts-a-un i molts-a-molts. Però en tots els exemples que hem vist fins ara tots els valors de la columna clau d'un dataframe apareixien a la columna clau de l'altre dataframe. Però això no sempre és així. De fet, és molt habitual que no ho sigui.

Vegem un exemple amb els dos dataframes següents que reflecteixen els menjars i begudes favorits d'alguns amics i provem de fer un join:

```
df1 = pd.DataFrame({'nom': ['Pere', 'Manel', 'Laura'],
'menjar': ['pasta', 'ensalada', 'pizza']})
df2 = pd.DataFrame({'nom': ['Pere', 'Jaume', 'Joana'],
'beguda': ['vi', 'cervesa', 'vi']})
df3 = pd.merge(df1,df2)
```

df1	nom	menjar
0	Pere	pasta
1	Manel	ensalada
2	Laura	pizza

df2	nom	beguda
0	Pere	vi
1	Jaume	cervesa
2	Joana	v

df3	nom	menjar	beguda
0	Pere	pasta	vi

Podem veure que en *df1* tenim el menjar favorit de Pere, Manel i Laura, mentre que en *df2* tenim la beguda favorita de Pere, Jaume i Joana. L'únic amic que està en els dos dataframes és Pere. Per això el join només retorna la informació d'ell.

De la mateixa manera que en l'àlgebra relacional, depenent de què facem amb els valors que estan en un costat i no en l'altre del join tendrem diversos tipus de joins. Ho podem especificar amb el paràmetre *how*, tal i com veurem a continuació.

Inner join (*how='inner'*)

Només prenem els valors que coincideixen en les dues columnes involucrades. És l'opció per defecte, la que hem vist a l'apartat anterior. També ho podem determinar de forma explícita amb *how='inner'*:

```
df3 = pd.merge(df1, df2, how='inner')
```

Left outer join (*how='left'*)

Prenem tots els valors de la columna de l'esquerra, però de la dreta només en prendrem els que coincideixin amb qualcun de l'esquerra. Ho especificam amb *how='left'*.

```
df3 = pd.merge(df1, df2, how='left')
```

El resultat és que apareixen els 3 amics de *df1*, però de *df2* només en surt Pere, l'únic que coincideix amb qualcun de *df1*.

	nom	menjar	beguda
0	Pere	pasta	vi
1	Manel	ensalada	NaN
2	Laura	pizza	NaN

Podem veure que el valor de la beguda favorita per a Manel i Laura és NaN (np.nan), no disponible.

Right outer join (*how='right'*)

Aquí agafam tots els valors de la columna de la dreta, mentre que de l'esquerra només n'agafarem els coincidents.

```
df3 = pd.merge(df1, df2, how='right')
```

	nom	menjar	beguda
0	Pere	pasta	vi
1	Jaume	NaN	cervesa
2	Joana	NaN	vi

Full outer join (*how='outer'*)

Prenem tots els valors, de l'esquerra i la dreta.

```
df3 = pd.merge(df1, df2, how='outer')
```

	nom	menjar	beguda
0	Pere	pasta	vi
1	Manel	ensalada	NaN
2	Laura	pizza	NaN
3	Jaume	NaN	cervesa
4	Joana	NaN	vi

9. Consultar un dataframe

Tot i que això no forma part de la fase de *data wrangling*, val la pena acabar aquest capítol veient com podem analitzar els datasets fent servir l'objecte *DataFrame* de la llibreria *pandas*. Veurem com fer queries, com ordenar els resultats, com emprar funcions d'agregració i com definir agrupacions.

9.1. Queries

El mètode *DataFrame.query()* es fa servir per consultar les files d'un dataframe segons una expressió (una condició sobre una o múltiples columnes), d'una manera semblant a SQL. El resultat és un nou dataframe.

Farem feina amb el següent dataframe:

```
df1 = pd.DataFrame({'nom': ['Joan', 'Aina', 'Maria', 'Pep'],
    'departament': ['Desenvolupament','Sistemes','Administració','Desenvolupament'],
    'anycontractacio': [2021, 2022, 2021, 2020] })
```

	nom	departament	anycontractacio
0	Joan	Desenvolupament	2021
1	Aina	Sistemes	2022
2	Maria	Administració	2021
3	Pep	Desenvolupament	2020

Suposem que volem recuperar les dades de Joan. Hem de cridar al mètode *query()* passant-li per paràmetre la condició *nom=='Joan'*:

```
df2 = df1.query("nom == 'Joan'")
```

	nom	departament	anycontractacio
0	Joan	Desenvolupament	2021

Si en lloc de fer servir un literal, tenim el valor en una variable, en la condició emprarem @ davant del nom de la variable:

```
empleat = 'Joan'
df2 = df1.query("nom == @empleat")
```

Podem utilitzar el paràmetre *inplace=True* per especificar que volem que el resultat de la query es guardi en el mateix dataframe. Hem d'anar molt alerta amb aquesta opció perquè això suposa que s'esborrin totes les files que no compleixin la condició.

```
df1.query("nom == 'Joan'", inplace=True)
```

Tornant al dataframe original *df1*, a més de l'operador == podem emprar qualsevol dels altres **operadors de comparació**: !=, >, >=, <, <=. També és freqüent utilitzar l'operador **in** per seleccionar els elements que estan dins d'un conjunt de valors. Per últim, també podem utilitzar **operadors lògics** per formar condicions múltiples: *and*, *or* i *not*. Vegem-ne un parell d'exemples.

Volem recuperar els empleats dels departaments de Sistemes i Desenvolupament:

```
df2 = df1.query("departament in ['Sistemes', 'Desenvolupament']")
```

I ara els que tenen un any de contractació major que 2020 i que són del departament de Desenvolupament:

```
df2 = df1.query("anycontractacio>2020 and departament=='Desenvolupament'")
```

En el cas que no vulguem recuperar totes les columnes, sinó només un subconjunt, podem emprar el nom de la columna o columnes. Per exemple, si només volem recuperar el nom:

```
df2 = df1.query("anycontractacio>2020 and departament=='Desenvolupament'")['nom']
```

O simplement:

```
df2 = df1.query("anycontractacio>2020 and departament=='Desenvolupament'")
df2 = df2['nom']
```

AMPLIACIÓ

Podeu veure els detalls del mètode *query()* a la documentació de *pandas*: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.query.html

Altres mètodes i propietats de l'objecte *pandas.DataFrame* útils per implementar consultes són:

- *pandas.DataFrame.filter()* per filtrar files per índex i columnes per nom
- *pandas.DataFrame.loc[]* per seleccionar files per etiquetes d'índex i columnes per nom
- *pandas.DataFrame.iloc[]* per seleccionar files per índex i columnes per posició (*deprecated* des de la versió 2.2.0)
- *pandas.DataFrame.apply()* per definir selects personalitzats emprant una funció lambda

9.2. Ordenacions

Els objectes *Series* i *DataFrame* ofereixen la possibilitat d'ordenar els valors mitjançant el mètode *sort_values()*.

En el cas de *DataFrame*, que és el que més ens interessa, aquest mètode té un paràmetre *by* on especificam la columna per la qual volem ordenar. Seguint amb l'exemple de l'apartat anterior:

```
df1.sort_values(by='nom')
```

Ordena les dades per la columna *nom*:

	nom	departament	anycontractacio
1	Aina	Sistemes	2022
0	Joan	Desenvolupament	2021
2	Maria	Administració	2021
3	Pep	Desenvolupament	2020

El mètode *sort_values()* també té un segon paràmetre *ascending* que ens permet espeficar l'ordre: ascendent (*True*) que és el valor per defecte o descendent (*False*). Per exemple:

```
df1.sort_values(by='anycontractacio', ascending=False)
```

Retorna les dades ordenades per any de contractació, de major a menor:

	nom	departament	anycontractacio
1	Aina	Sistemes	2022
0	Joan	Desenvolupament	2021
2	Maria	Administració	2021
3	Pep	Desenvolupament	2020

9.3. Agregacions

Els objectes *Series* i *DataFrame* ofereixen un conjunt de funcions d'agregació que són molt útils a l'hora d'analitzar les dades. Estan implementades mitjançant aquests mètodes:

Funció d'agregació	Descripció
count()	Número de files no nul·les (diferents de *np.nan*)
sum()	Suma de tots els valors no nuls
min(), max()	Mínim i màxim de tots els valors no nuls
first(), last()	Primer i darrer valor no nul
mean(), median()	Mitjana i mediana de tots els valors no nuls
std(), var()	Desviació estàndard i variància de tots els valors no nuls
prod()	Producte de tots els valors no nuls

Taula: *Funcions d'agregació de pandas.*

Vegem algun exemple, amb el següent dataframe:

```
import pandas as pd
df1 = pd.DataFrame({'nom': ['Joan', 'Aina', 'Maria', 'Pep'],
    'departament': ['Desenvolupament','Sistemes','Administració','Desenvolupament'],
    'anycontractacio': [2021, 2022, 2021, 2020],
    'salari': [30000, 35000, 25000, 40000] })
```

	nom	departament	anycontractacio	salari
0	Joan	Desenvolupament	2021	30000
1	Aina	Sistemes	2022	35000
2	Maria	Administració	2021	25000
3	Pep	Desenvolupament	2020	40000

Si no especificam cap columna, les funcions d'agregració ens retornaran el valor corresponent de cada una d'elles. Per exemple:

```
df1.max()
```

Ens retorna el valor més gran de cada columna:

```
nom                   Pep
departament       Sistemes
anycontractacio       2022
salari               40000
dtype: object
```

En canvi, podem aplicar la funció a una única columna:

```
df1['salari'].sum()
```

Que ens retorna

```
130000
```

Una altra manera habitual d'emprar les funcions d'agregació és mitjançant el mètode *aggregate()*, o el seu alias *agg()*, dels objectes *Series* i *DataFrame*. A aquest mètode li passam per paràmetre la funció que volem emprar (de la taula que hem vist abans). Per exemple:

```
df1.agg('max')
```

Ens retorna l'equivalent a *df1.max()*:

```
nom                    Pep
departament        Sistemes
anycontractacio        2022
salari                40000
dtype: object
```

Un avantatge del mètode *aggregate()* (o *agg()*) és que ens permet emprar altres funcions per fer l'agregació. Fins i tot en podríem definir la nostra personalitzada. En aquest exemple, fa servir *np.mean* per fer l'agregació:

```
df1['salari'].agg(np.mean)
```

El mètode *aggregate()* (o *agg()*) també permet aplicar d'una vegada diverses funcions d'agregació. Per exemple:

```
df1['salari'].agg(['max', 'min', 'mean'])
```

Que dona com a resultat:

```
max     40000.0
min     25000.0
mean    32500.0
Name: salari, dtype: float64
```

9.4. Agrupacions

Molt sovint quan analitzam les dades, ens interessa primer agrupar-les per alguna (o algunes) de les columnes. Per exemple, seguint amb del dataframe de l'apartat anterior, és possible que vulguem saber quin és el salari total de cada departament. Per fer una agrupació hem d'emprar el mètode *groupby()*:

```
df1.groupby('departament')
```

Ens retorna un objecte del tipus *GroupBy*. Aquest objecte és molt semblant a *DataFrame* i té també els mètodes per aplicar les funcions d'agregació que hem vist abans.

Així doncs, per recuperar el salari mitjà de cada departament, una vegada feta l'agrupació, primer prenem només la columna *salari* i després cridam al mètode *sum()*:

```
df1.groupby('departament')['salari'].sum()
```

Que ens retorna:

```
departament
Administració      25000
Desenvolupament    70000
Sistemes           35000
Name: salari, dtype: int64
```

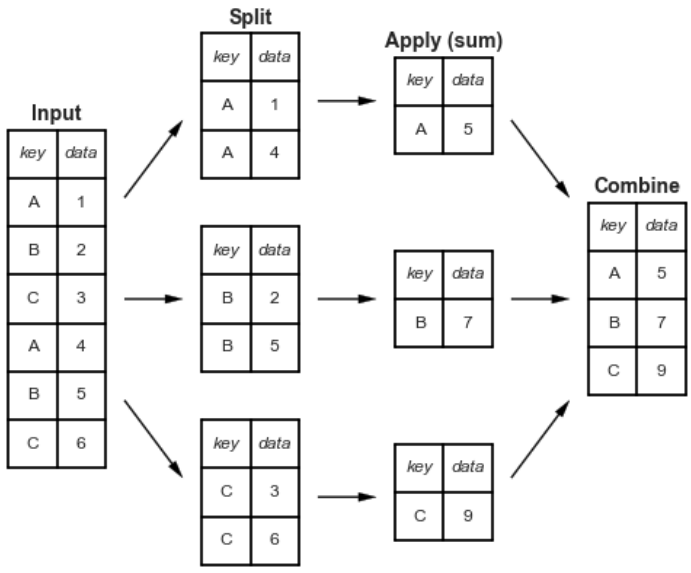
Fixau-vos que, per defecte, es mostren els resultats ordenats per la columna d'agrupació, en ordre creixent.

Aquest és un exemple bastant bàsic d'agrupació-agregació, tot i que n'és el més habitual.

El procés d'agrupació-agregació segueix 3 fases:

- *split*, on tots les files que tenen el mateix valor de la columna clau s'agrupen juntes
- *apply*, on s'aplica la funció d'agregació
- *combine*, on es combinen els valors obtinguts en la fase anterior per generar la resposta

La següent imatge ho mostra gràficament:



Imatge: *Fases del procés d'agrupació-agregació. Font: Python Data Science Handbook, de Jake VanderPlas*

Podríem aprofundir en cada una d'aquestes fases, però ja queda fora de l'objecte d'aquest curs.

Si hi estàs interessat, pots consultar el llibre *Python Data Science Handbook*, de Jake VanderPlas, o consultar algun tutorial, com el de RealPython.

10. Bibliografia

Per elaborar aquests apunts, ens hem basat parcialment en els següents llibres sobre ciència de dades amb Python. Podeu trobar més detalls sobre tot el que hem vist aquí a qualsevol d'ells, tots tres són excel·lents:

- *Python Data Science Handbook*, per Jake VanderPlas. O'Reilly Media (2016). Aquest llibre està disponible en obert, sota llicència CC-BY-NC-ND, a https://jakevdp.github.io/PythonDataScienceHandbook/
- *Python for Data Analysis*, 2a edició, per Wes McKinney. O'Reilly Media (2017). Podeu trobar el codi Python a https://github.com/wesm/pydata-book
- *Data Science from Scratch*, 2a edició, per Joel Grus. O'Reilly Media (2019). Podeu trobar el codi Python a https://github.com/joelgrus/data-science-from-scratch