Resolución examen PED febrero 2020

TEST

1. ¿Cuál es el resultado de ejecutar el siguiente código?

```
def fun(a=2,b=3):
    return a*b
print(fun(b=5))
```

Solución: Se imprime el valor 10

Explicación:

En la cabecera de la función se definen los parámetros y sus valores por defecto: 2 para el primer parámetro llamado a y 3 para el segundo parámetro llamado b. Cuando se llame a la función fun sin declarar los valores de dichos parámetros, se tomarán esos valores por defecto.

La llamada fun(b=5) establece un valor de 5 para el parámetro b (aunque sea el segundo), mientras que el parámetro a toma su valor por defecto ya que no se ha declarado un nuevo valor en esta llamada.

En consecuencia, los valores de los parámetros a y b recibidos por la función son 2 (por defecto) y 5 (valor establecido en la llamada) y el resultado devuelto es, lógicamente, 10.

2. Dado el siguiente fragmento de código:

```
import pandas as pd
import numpy as np
data = pd.DataFrame(np.arange(20).reshape((5, 4)),
index = ['X', 'Y', 'Z', 'W','V'],
columns = ['X', 'Y', 'Z', 'W'])
```

¿Qué ocurre si ejecutamos data=data.drop('Y')?

Solución: Se elimina la fila Y de data

Explicación:

Al ejecutar

```
data = pd.DataFrame(np.arange(20).reshape((5, 4)),
index = ['X', 'Y', 'Z', 'W','V'],
columns = ['X', 'Y', 'Z', 'W'])
```

El resultado es un dataFrame con el siguiente aspecto:

	X	**Y**	**Z**	**W**
X	0	1	2	3
Y	4	5	6	7
Z	8	9	10	11
W	12	13	14	15
V	16	17	18	19

Esto se debe a que el comando np.arange(20) nos crea un array de 20 elementos, empezando desde 0 y terminando en 19. Con el comando reshape((5, 4)), convertimos ese array en uno de dos dimensiones con 5 filas y 4 columnas. Es este array de dos dimensiones el que define el formato del dataFrame. Posteriormente, indicamos que las filas del dataFrame se denominan 'X', 'Y', 'Z', 'W' y 'V', mientras que las columnas se denominan 'X', 'Y', 'Z' y 'W'.

El comando drop sirve para eliminar filas o columnas. A través del atributo axis se selecciona la dimensión a eliminar. Con axis=0 se eliminan filas, y con axis=1 se eliminan columnas. Por defecto axis vale 0. Por tanto, data.drop('Y') elimina la fila Y. Si el comando fuera data.drop('Y',axis=1), eliminaría la columna Y.

3. Dado el siguiente fragmento de código, cuál será el contenido de l tras ejecutarse el código:

```
a = (x ** 2 for x in range(10) if x % 2 == 0)
l = [x for x in a if x % 3 == 0]
```

Solución: [0, 36]

Explicación:

El primer comando crea un generador de las potencias de 2 de los números pares del 1 al 10. Este generador, si se le fuera llamando, iría devolviendo los siguientes valores: 0, 4, 16, 36, 64.

Posteriormente, en l creamos una lista por comprensión donde solo nos quedamos con los múltiplos de 3 de los valores que devuelve el generador a. Por tanto, solo nos quedamos con 0 y 36, dando lugar a la lista [0, 36].

4. ¿Cuál es el resultado de ejecutar el siguiente código?

```
tup = (1, [] , "a")
tup[1].append(4)
print (tup[1])
```

Solución: Se imprime [4]

Explicación:

La variable tup contiene una tupla, que es un tipo de datos inmutable, por lo que no se puede modificar. Sin embargo, su segundo componente (al que se accede mediante tup[1]) es una lista, que sí es un tipo de datos mutable.

Al realizar tup[1].append(4) se está modificando en realidad la lista que es el segundo componente de la tupla tup, por lo que la lista, originalmente vacía, pasa a valer [4] y ese es el valor que se imprime.

Incluso, si la última instrucción fuera:

```
print (tup)
```

el resultado sería (1, [4], "a") porque hemos modificado un dato mutable aunque esté dentro de otro inmutable.

PROBLEMAS

Problema 1. El director de una empresa está interesado en contratarnos como científicos de datos y en la entrevista nos informa de que tienen un conjunto de datos guardado en una hoja de cálculo con el que quieren trabajar.

El director nos informa de que en primer lugar solo quieren trabajar con datos numéricos. Además, como su conjunto de datos contiene muchas filas y su empresa tiene recursos de computación limitados, nos pide que si podemos hacer algo para que ocupe lo mínimo posible en memoria.

Para realizar el ejercicio se pide explicar a alto nivel qué acciones tomaríamos para realizar lo que nos pide el director. Posteriormente, se pide escribir el código asociado a ello. Finalmente, añadir código para ver qué tamaño (en cuanto a filas y columnas) tenía la colección inicial y qué tamaño tenía la colección final.

Solución:

Nota: existen muchas soluciones posibles, ya que el ejercicio pide trabajar a alto nivel y no describe completamente los datos concretos. Por este motivo, se van a dar orientaciones al respecto, con su código asociado. Es muy importante indicar toda suposición que se tenga sobre los datos.

En primer lugar, nos informan de que los datos están contenidos en una hoja de cálculo, pero no sabemos si lo tienen directamente en formato Excel o lo han exportado a formato csv o similar. En cualquier caso, el primer paso sería realizar la importación de forma adecuada según el formato, para tener un dataFrame que podamos explorar.

Asumimos que podemos cargar el fichero entero en memoria. El código asociado sería:

```
import pandas as pd
#importación a partir de un fichero csv:
datos = pd.read_csv('datos.csv')
# importación de Excel asumiendo que los datos
# están en la hoja denominada Hoja1:
datos = pd.read_excel('datosHojaCalculo.xlsx', 'Hoja1')
```

Lo primero es explorar el dataFrame para recopilar información al respecto. Para ello vemos el contenido de algunas filas (head) y obtenemos

información del propio dataset una vez cargado (info y describe, viendo el espacio que ocupa en memoria).

```
print(datos.head())
print(datos.tail())
print(datos.info())
print(datos.describe())
```

Guardamos información sobre el número de columnas y de filas.

```
numFilasInicial = len(datos)
numColumnasInicial = len(datos.colums)
```

Nos quedamos solo con las columnas numéricas del dataset tal y como nos piden en el enunciado. Lo más eficiente es indicar los posibles tipos numéricos y, usando la función *select_dtypes*, quedarnos solo con esas columnas. Pero si no conocemos dicha función, podemos hacer la selección de forma manual indicando las columnas concretas que queremos (tras obtener la información al explorar los datos). A continuación incluímos el código de los dos métodos:

```
Datos = datos.select_dtypes(include="number")
#indicamos una lista con las columnas numéricas.
# Aquí solo hemos puesto las de nombres col1 y colX,
# pero podrían ser más
datos = datos[[“col1”,”colX”]]
```

Volvemos a explorar el dataset para ver el resultado y ahora incluimos el atributo memory_usage con el valor deep para obtener información más precisa sobre el espacio que ocupan nuestros datos en memoria.

```
print(datos.info(memory_usage="deep"))
```

A continuación, comenzamos a reducir el tamaño del dataset. Para ello, primero ajustamos los tipos utilizados a aquellos que nos permitan guardar la misma información en menos espacio. Reducimos primero el espacio de las columnas que contienen enteros y a continuación las que contienen números en coma flotante (se puede poner todo en un mismo bucle, pero lo separamos por claridad)

```
# reducción de espacio de columnas con enteros
for col in datos.columns:
    if "int" in str(datos[col].dtype):
        datos [col] =
            pd.to_numeric(datos[col],downcast='unsigned')

# reducción de espacio de columnas con números
# en coma flotante
for col in datos.columns:
    if "float" in str(datos [col].dtype):
        datos [col] =
            pd.to_numeric(datos [col],downcast='float')
```

Podríamos explorar si en alguna columna la cantidad de valores distintos es muy baja. Pero estos datos estarán ya representados como enteros, o float, que ocupen poco espacio, por lo que no sería necesario cambiar el tipo a categoría. Sí que tiene sentido hacerlo cuando los datos son de tipo object, que ocupan más. Pero como solo nos hemos quedado con columnas numéricas, no es el caso.

Otra posibilidad para reducir espacio podría ser eliminar filas repetidas. Sin embargo, es una decisión que no podemos tomar unilateralmente porque la empresa podría querer mantener todas estas filas. Por tanto, tendríamos que preguntar al respecto. Si nos dicen que prefieren borrarlas, lo hacemos y vemos la diferencia en espacio.

```
datos. drop_duplicates(inplace = True)
print(datos.info(memory_usage="deep"))
```

Ahora vemos el tamaño final del dataFrame:

```
numFilasFinal = len(datos)
numColumnasFinal = len(datos.colums)
print("Inicialmente teníamos",numFilasInicial,
      "filas y ahora tenemos",numFilasFinal)
print("Inicialmente teníamos", numColumnasInicial,
      "columnas y ahora tenemos", numColumnasFinal)
```

Problema 2. Un portal de internet que ofrece una aplicación similar a spotify (streaming de música) nos contrata como expertos en datos para ayudarles a gestionar su base de datos de canciones. Esta base de datos se encuentra almacenada en un fichero CSV con la siguiente estructura:

- En cada línea se encuentran los campos de cada canción, separados por comas.
- En la primera línea se encuentran los nombres de los campos, separados por comas, a modo de cabecera.
- De cada canción se dispone de los siguientes datos: nombre, grupo, género y duración. Los tres primeros almacenados como texto y el último almacenado como un entero que indica el número de segundos de duración de la canción.

También nos indican que, tras analizar el comportamiento de sus usuarios, las tres acciones más demandadas son las siguientes:

a) Crear una lista de canciones con la discografía completa de un grupo (todas las canciones de dicho grupo).
b) Crear una lista aleatoria de canciones de un determinado género que, conjuntamente, tenga una duración aproximada de X segundos. Es decir, que si al añadir una canción llegamos a X segundos en total (o nos pasamos), dejamos de añadir canciones a la lista.
c) Encontrar la canción más larga de un determinado género.

Para realizar el ejercicio se pide explicar a alto nivel qué acciones tomaríamos para preparar los datos de manera que estas operaciones se ejecuten en el menor tiempo posible y, adicionalmente, escribir el código asociado a ello. Por último, se pide el código necesario para resolver las tres acciones que más demandan los usuarios.

Solución:

Al igual que en el ejercicio anterior, asumimos que podemos cargar el fichero entero en memoria. El código para realizar esto sería:

```
import pandas as pd
can = pd.read_csv('canciones.csv')
```

ya que sabemos que los datos de las canciones están almacenados en un fichero CSV. No necesitaríamos especificar más parámetros, ya que sus valores por defecto nos valdrían.

Una vez cargado el CSV, podemos proceder a optimizarlo. Analicemos cada una de las columnas que tenemos:

1. El primer dato de que disponemos es el nombre de la canción. Parece complicado que esta columna pueda ser optimizada (por ejemplo, convirtiéndola a categórica) por la gran diversidad de títulos que es esperable encontrar. Así pues, la primera columna no la vamos a tocar.

2. La segunda columna sería el nombre del grupo (asumamos que está etiquetado como "`Grupo`"). Podríamos pensar en convertir esta columna a categórica:

   ```
   can['Grupo']=can.Grupo.astype('category')
   ```

   pero deberíamos comprobar el número diferente de grupos que existen, ya que si hubiera muchos grupos, esta supuesta mejora podría ser contraproducente. Para eso, podríamos utilizar:

   ```
   print (len(can.Grupo.unique()))
   ```

   para saber cuántos grupos diferentes existen y así poder decidir, comparando el número de grupos con el número total de canciones, si convertir a categórica esta columna o no.

3. La tercera columna (que asumiremos etiquetada como "`Genero`") sí que parece mucho más susceptible de ser convertida a categórica, pues el género es un dato que (con mucha probabilidad) se repetirá mucho y al tratarlo como categórico el uso de memoria se debería reducir bastante:

   ```
   can['Genero']=can.Genero.astype('category')
   ```

4. Por último, la duración de las canciones (etiquetada como "`Duracion`") está representada como un entero. Python lo tratará como un entero de 64 bits (`int64`), pero conociendo qué se almacena aquí, es posible reducir el tamaño de los enteros utilizados.

   En primer lugar, sabemos que la duración de una canción es un dato que nunca va a ser negativo, por lo que usaremos uno de los tipos sin signo. No podríamos reducirlo a `uint8`, porque es muy probable que nos encontremos con una canción de más de 255 segundos (4 minutos y 15 segundos), por lo que sería necesario reducirlo a `uint16` (lo que nos da un rango de hasta 65535 segundos, que son más de 18 horas):

```
can['Duracion']=can.Duracion.astype('uint16')
```

Con esta acción hemos reducido en 6 bytes la memoria necesaria para almacenar la duración de cada canción.

Tras realizar todas estas optimizaciones, los datos deberían ocupar menos memoria. Esto lo podremos comprobar ejecutando:

```
print (can.info(memory_usage='deep'))
```

antes y después de las optimizaciones, para así poder ver la mejora de tamaño tras optimizar. Una vez cargado y optimizado el fichero, pasamos a resolver las tres operaciones más demandadas por los usuarios:

a) **Crear una lista de canciones con la discografía completa de un grupo (todas las canciones de dicho grupo).**

En primer lugar, vamos a crear la cabecera de la función, que recibirá un parámetro con el nombre del grupo:

```
def discografia(grupo):
```

Ahora crearemos un filtro para obtener todas las filas que se correspondan con canciones de ese grupo. Así:

```
can['Grupo'] == grupo
```

nos genera un dataframe con una única columna de tipo `bool` en la que tendremos los valores True si la misma fila de nuestro dataframe can contiene una canción del grupo buscado y False en caso contrario.

Ahora, necesitamos un dataframe con la información contenida en las filas en las que estamos interesados:

```
can[can['Grupo'] == grupo]]
```

Pero sólo nos interesa la columna de los títulos de las canciones:

```
can[can['Grupo'] == grupo]['Nombre']
```

Y, además, nos interesa convertida en una lista:

```
can[can['Grupo'] == grupo]['Nombre'].tolist()
```

Por lo que el código completo de la función sería:

```
def discografia(grupo):
  return can[can['Grupo']==grupo]['Nombre'].tolist()
```

b) **Crear una lista aleatoria de canciones de un determinado género que, conjuntamente, tenga una duración aproximada de X segundos. Es decir, que si al añadir una canción llegamos a X segundos en total (o nos pasamos), dejamos de añadir canciones a la lista.**

Al igual que en la pregunta anterior, definimos la cabecera de la función que nos va a devolver la lista aleatoria de canciones:

```
def listaAleatoriaGenero(genero,durmax):
```

donde el parámetro `genero` será el género en el que estamos interesados y el parámetro `durmax` la duración máxima.

Y, del mismo modo, filtramos las filas que contienen canciones del género pedido:

```
can[can['Genero']==genero]
```

Y ahora generamos un dataframe auxiliar con las filas desordenadas. Para eso podemos utilizar el método sample:

```
aux = can[can['Genero']==genero].sample(frac=1)
```

Con ese código estamos generando una reordenación aleatoria de todas las filas de nuestro dataframe que contienen canciones del género buscado. Nótese que se especifica que el muestreo aleatorio que se quiere obtener cubre el 100% (`frac=1`) de nuestro dataframe original.

Ahora ya podemos iterar sobre ese nuevo dataframe para ir seleccionando canciones mientras aún podamos:

```
duracion = 0
lista = []
for row in aux.itertuples():
  duracion = duracion + row.Duracion
  lista.append(row.Nombre)
  if duracion >= durmax: break
```

Así, vamos añadiendo canciones a una lista mientras la duración total acumulada sea menor que la duración máxima. En el momento en el que se alcanza dicha duración máxima, se corta el bucle y en la variable `lista` tenemos el resultado a devolver.

Por lo tanto, el código completo de la función sería:

```
def listaAleatoriaGenero(genero,durmax):
  aux = can[can['Genero']==genero].sample(frac=1)
  duracion = 0
  lista = []
  for row in aux.itertuples():
    duracion = duracion + row.Duracion
    lista.append(row.Nombre)
    if duracion >= durmax: break
  return (lista)
```

c) **Encontrar la canción más larga de un determinado género.**

Vamos a comenzar como siempre. Definimos la cabecera de la función:

```
def cancionMasLargaGenero(genero):
```

Y nos quedamos con todas las filas que tengan las canciones del género buscado:

```
can[can['Genero']==genero]
```

Ahora obtenemos el índice de la fila que contiene el valor más alto en la columna de la duración:

```
ir=can[can['Genero']==genero]['Duracion'].idxmax()
```

Y ya sólo nos queda obtener el nombre de esa canción:

```
can.iloc[ir].Nombre
```

Con lo que el código completo sería:

```
def cancionMasLargaGenero(genero):
  ir=can[can['Genero']==genero]['Duracion'].idxmax()
  return (can.iloc[ir].Nombre)
```