PRACTICAS CURSO DE PYTHON AVANZADO

REPASO OBJETOS Y TIPOS:

1) Colecciones: Se piden dos números, uno menor que 10 y otro mayor que 100. El primer número representa el número de intervalos y hay que calcular los rangos de los intervalos según el segundo número. Por ejemplo: n1 = 4, n2 = 100. Los rangos serían de 0..24, 25..49, 50..74, 75..99.

Después se generan números al azar que no sobrepasen el número n2 y se almacenan en cada intervalo. Luego listar los intervalos con los números que se han recogido. Hay que clasificar los números en el intervalo adecuado.

2) Objetos: Implementar una clase i18n que sirva para implementar la internacionalización de aplicación. En la carpeta: practicas/colecciones_objetos hay dos ficheros para español y otro para inglés. La clase tiene que controlar excepciones, por ejemplo, si no encuentra el fichero del idioma. Los ficheros tienen un formato: clave=valor

El usuario para utilizarla hará:

idioma = Idioma('en')

idioma.getString('clave')

Se puede considerar también un posible cambio de idioma.

THREADS:

1) Implementar el esquema del productor - consumidor con **semáforos y mutex** para N productores y M consumidores. Después de probarlo se puede escribir con una instrucción **with** cuando se intente leer o escribir en el buffer común.

2) Idem del anterior pero con una **queue**.

3) Tenemos que **descargar 100 ficheros**, cada fichero tiene un aleatorio entre 0 y 99999.

Los ficheros se encuentran en la URL:

http://www.dpii.es/files/fich0.txt
http://www.dpii.es/files/fich1.txt
…
…
http://www.dpii.es/files/fich98.txt
http://www.dpii.es/files/fich99.txt

Diseñar una clase **Lanzadera** que recibe el número de particiones a realizar. Se trata de dividir la descarga de los 100 en n particiones. Por ejemplo, si n es 5, se crearán 5 hilos y cada hilo descargará (en este caso), 20 ficheros cada uno, con el siguiente reparto: el hilo 1 → del 0 al 19, el 2 → del 20 al 39 … , 5 → del 80 al 99

Los ficheros no hay que grabarlos, recuperar el número y almacenarlo en una estructura "**compartida**", teniendo en cuenta que habrá otro hilo con un comportamiento distinto que irá tomando los números que descargan los otros hilos y los guardará en una lista. Cuando se hayan descargado los 100 ficheros, este último hilo ordenará los resultados y los grabará en un fichero de texto. *La aplicación puede imprimir mensajes donde se vea la URL que se esta descargando y el número.*

El código principal del programa:

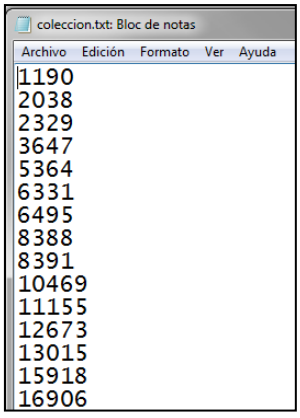
```
if __name__=='__main__':
        lanzadera = Lanzadera(5)
        lanzadera.lanzarHilos()
```

El 5 se puede leer de teclado, pero se debería de cumplir **100 % n == 0**

Para grabar un fichero:

```
fich = open("coleccion.txt","w")
fich.write(str(numero)+"\n")
fich.close()
```

El fichero resultante quedará:



En la carpeta de **practicas/threads** se encuentra el fichero: **descarga_url.py** que implementa una función que se le envía una URL accede y recupera el número de este fichero.

PATRONES DE DISEÑO:

1) Aplicar el patrón **SINGLETON** al clase i18n que solo se crea una instancia. Utilizar las distintas formas de implementar el Singleton en Python.

2) Aplicar el patrón **ABSTRACTFACTORY** en la siguiente situación: Tenemos 3 fabricantes distintos (Samsung, HTC, Nokia) y dos tipos de productos (SmartPhones y Tabletas). Los 3 fabricantes fabrican ambos tipos de productos cada uno con sus modelos concretos.

- Samsung fabrica SmartPhone (Galaxy S7) y Tablet (Guru).
- HTC fabrica SmartPhone (Titan) y Tablet (Genie)
- Nokia fabrica SmartPhone (Lumia) y Tablet (Asha)

Identificar la factoría abstracta, las factorías concretas y los productos (abstractos y concretos).

Las características que nos interesan los productos son: Sistema operativo, memoria RAM, memoria interna y nombre del producto.

Seleccionar una factoría, de tal forma que cuando le pidamos un SmartPhone o una Tablet nos devuelva los de factoría concreta.

La Factoría genérica dispone de un método static para crear una factoría concreta elegida por el usuario. Utilizar el decorador **@staticmethod** para hacer el método static.

Ejemplo para Samsung

```
Indica el fabricante:
Samsung
Fabricante:
GalaxyS7: SmartPhone: Android 7.0, 2, 10, galaxyS7
Guru: Tablet: Android 5.0, 3, 4, guru
```

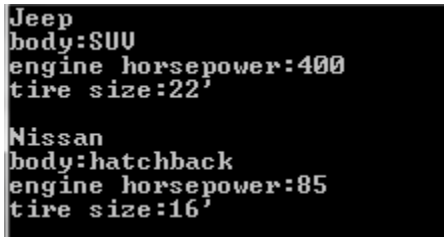
USO DE LAS CLASES (CÓDIGO PRINCIPAL)

- # Seleccionar la factoría y construir productos concretos de esta:
- print "Indica el fabricante:"
- fab = input()
- fabricante = Factoria.createFactoria(fab)
- sm = fabricante.getSmartPhone()
- tablet = fabricante.getTablet()
- print ("Fabricante:",fab)
- print (sm)
- print(tablet)

3) Aplicando el patrón **PROTOTYPE** (ver ejemplo)

- Implementar la paleta de componentes de distintas figuras gráficas. Las posibles figuras pueden ser: Circulo, Rectángulo y Triángulo.
- El prototipo puede ser una clase abstracta con el método clone() abstracto, para que sea obligatorio implementarlo en las subclases.
- import abc
- Heredar de **abc**.**ABC** y también proporciona el decorador: **@abc.abstractmethod**

4) Aplicar el patrón **BUILDER** para construir dos tipos de coche. El coche se construye por partes: Tamaño de la rueda, Ruedas, Chasis, Motor.



Tenemos un producto final: EL **COCHE**

El coche se desglosa en 3 PARTES FUNCIONALES: **RUEDAS**, **CHASIS** y **MOTOR**

Otra clase es el **DIRECTOR**: Es el que se encarga de ensamblar las piezas. Tiene asociado un Builder **que se inyecta desde fuera**. **El Director debería de tener un atributo del Builder que va a utilizar para la construcción de las distintas partes del producto final** (el coche).

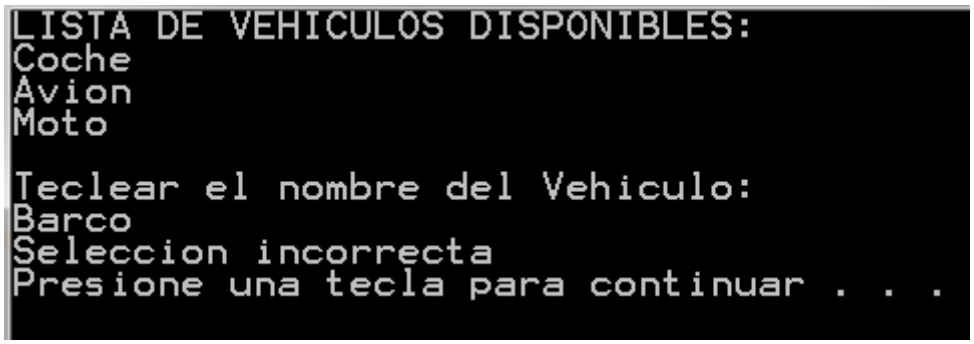
Podemos tener distintos **FABRICANTES de COCHE** estos son los **BUILDER** (los constructores, específicos). La construcción de las piezas depende de cada constructor.

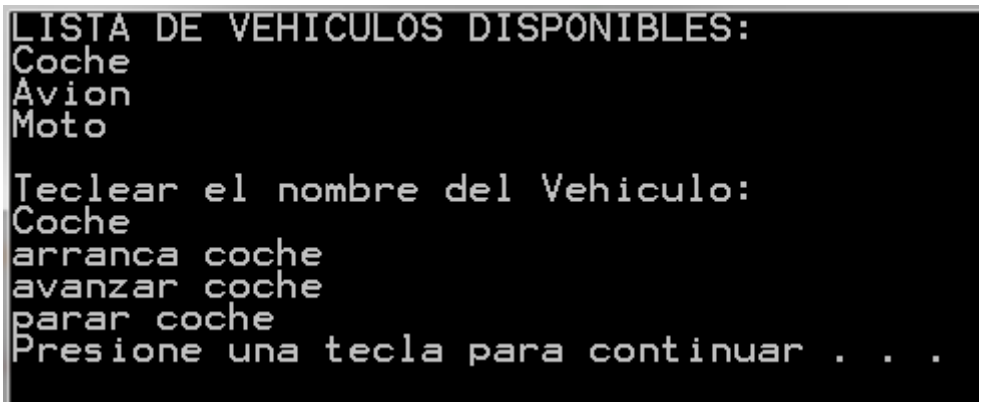
Cada Builder tiene que suministrar la forma de CONSTRUIR CADA PARTE DEL PRODUCTO FINAL.

De esta forma es sencillo añadir NUEVOS FABRICANTES. Para ello tendremos una clase Abstracta Builder que la base para cada Builder específico de un Fabricante. Tendrá un método para construir cada parte del coche.

5) Aplicando el patrón **FACTORY METHOD** disponemos de una serie de vehículos: La clase abstracta vehículo proporciona 3 métodos: arrancar, avanzar, parar, etc. Y cada vehículo tiene esos 3 métodos pero en cada clase el comportamiento es distinto.

Ejemplo de funcionamiento:





El menú se genera de forma dinámica utilizando el método: **IVehiculo.__subclasses__()**

La factoría para seleccionar la clase a instanciar lo puede hacer una forma general con la función eval() → return eval("{}()".format('Coche'))

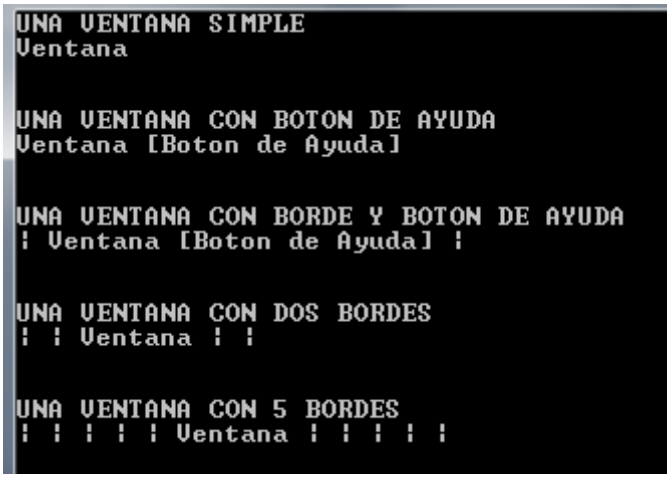
VER EJEMPLOS FACTORIAS POLIMORFICAS → DONDE CADA PRODUCTO TIENE SU PROPIA FACTORIA, Y LA CLASE FACTORIA ALMACENA LAS DISTINTAS FACTORIAS.

6) **PATRÓN FACHADA** Implementar este patrón para simular la concesión de una Hipoteca. El proceso para el cliente es el siguiente: hay que comprobar si en el banco hay saldo por encima de un valor. En el sistema de créditos hay que comprobar si hay suficiente crédito y comprobar que el cliente no tenga impagos en el servicio de impagos.

7) Utilizando el patrón ADAPTER dar dos soluciones a la clase VectorPlano (***por Composición y herencia múltiple***) que tiene que implementar la interfaz proporcionada: Vector2D y disponemos de la clase Vector3D (que no se puede cambiar) hay que adaptarla para que la nueva clase VectorPlano se adapte a la interfaz Vector2D utilizando la clase Vector3D.

8) Utilizando el patrón DECORATOR implementar la posibilidad de implementar Ventanas normales (imprimen un mensaje por pantalla), VentanasConBordes (imprimen el mensaje con borde **| Ventana |**), con Botón de Ayuda, pero tener en

cuenta que queremos tener la posibilidad de añadir más de un borde si queremos, y entonces veríamos algo así : **||| Ventana |||**.



Jerarquía de clases:

IVentanaAbstracta hereda de object y sólo tiene un método dibujar (a implementar por las subclases).

Ventana: Hereda de IVentanaAbstracta (representa la ventana simple), sobrescribe el método dibujar.

VentanaDecorator: Hereda de IVentanaAbstracta. De esta clase heredan todos los decoradores. Tendremos un decorador que pone el borde y otro el botón de ayuda. Mirar el esquema de UML (de la documentación), tiene un atributo de la VentanaAbstracta (IVentanaAbstracta en nuestro caso). Sobrescribe el método dibujar.

BordeDecorator: Hereda de VentanaDecorator, (en dibujar pinta los bordes y algo más).

BotonDeAyudaDecorator: Hereda de VentanaDecorator, (en dibujar pinta el botón de ayuda y algo más).

9) Utilizando el patrón CADENA DE RESPONSABILIDAD. Disponemos de 3 Widgets (MainWindow, SendDialog, MsgText) cada Widget dispone de uno o varios eventos.

MainWindow (close y default), SendDialog (paint) y MsgText (down).

Si tenemos la siguiente Cadena de Responsabilidad:

- mw = MainWindow()
- sd = SendDialog(mw)

- msg = MsgText(sd)
- *Según el código el padre de MsgText es SendDialog y el padre de este último es MainWindow.*

Y mandamos un **evento** (una clase simple que almacena el nombre del evento y el método __str__). Cada Widget tendrá que comprobar si tiene el o no, ese evento, si lo tiene, lo atiende (imprime un mensaje y el evento), y si no lo pasa al Widget padre.

El Widget es la superclase de (MainWindow, SendDialog, MsgText) dispone de método handle que recibe un evento y determina si un objeto lo tiene o no (si lo tiene lo atiende, y si no lo pasa al objeto padre.)

Así podemos determinar si un objeto tiene o no un método:

```
if hasattr(self, handler):

    # Si el metodo lo tiene el, lo atiende:

    method = getattr(self, handler)

    method(event)
```

10) Utilizando el fichero: **practicas/patrones/comportamiento/ordenamiento.py**

Aplicar el patrón ESTRATEGIA con los 2 métodos de ordenación proporcionados.

ANALISIS DE DATOS: NUMPY

Ejercicios:

- Crear un vector con valores dentro del rango 10 a 49
- Invertir un vector
- Crear una matriz de 3 x 3 con valores de 0 a 8.
- Encontrar los índices que no son ceros del arreglo [1,2,4,2,4,0,1,0,0,0,12,4,5,6,7,0]
- Crear una matriz identidad de 6x6
- Crear una matriz con valores al azar con forma 3x3x3
- Encontrar los índices de los valores mínimos y máximos de la anterior matriz
- Crear una matriz de 10x10 con 1's en los bordes y 0 en el interior (con rangos de índices)
- Crear una matriz de 5 x 5 con valores en los renglones que vayan de 0 a 4.
- Crear dos arrays al azar y comprobar si son iguales.

1. Generar dos **np.array** con número aleatorios (import random o utilizar el módulo random de numpy) y para generar (random.randint(valor1, valor2) guardarlos en dos archivos, recuperarlos y realizar operaciones con ellos … También se pueden generar valores random con np.random.randn(6,3) → Genera una matriz de 6 x 3

ANALISIS DE DATOS: PANDAS

1) Series: generar una serie con las letras del abecedario como claves y como valores utilizar números aleatorios.

2) Implementar una clase que le indiquemos los parámetros de la conexión y una tabla y nos devuelva un DataFrame cargado y con los nombres de las columnas de la tabla. Implementar __init__ y getDataFrame('nombre_tabla')

3) Reutilizar esta clase para crear otra, y sobrescribir el método getDataFrame que utilice el módulo pandas.io.sql

4) A partir de los ficheros **practicas/Analisis Datos/csv**, generar un script que recorra los ficheros **csv** aquí contenidos y genere un fichero **xlsx**. El contenido de los ficheros es de la evolución de huracanes. En los ficheros Excel se copiarán todas las columnas y en el caso de las filas sólo el primer registro de cada cambio de estado que tenga el huracán, por ejemplo: para Irma.

	A	B	C	D	E	F	G	H	I
		Date	Time	Lat	Lon	Wind	Pressure	Storm Type	Category
0		Aug 30	15:00 GMT	16.4°	-30.3°	50 mph	1004 mb	Tropical Storm	-
1		Aug 31	15:00 GMT	16.9°	-33.8°	100 mph	979 mb	Hurricane	2
2		Aug 31	21:00 GMT	17.3°	-34.8°	115 mph	967 mb	Hurricane	3
3		Sep 01	15:00 GMT	18.5°	-37.8°	110 mph	972 mb	Hurricane	2
4		Sep 01	21:00 GMT	18.8°	-39.1°	120 mph	964 mb	Hurricane	3
5		Sep 02	09:00 GMT	19.0°	-41.8°	110 mph	970 mb	Hurricane	2
6		Sep 03	09:00 GMT	18.0°	-47.5°	115 mph	969 mb	Hurricane	3
7		Sep 04	21:00 GMT	16.7°	-54.4°	130 mph	944 mb	Hurricane	4
8		Sep 05	11:45 GMT	16.7°	-57.7°	175 mph	929 mb	Hurricane	5

El último registro también nos interesa.

Implementar una clase que reciba la carpeta donde se encuentran los ficheros, la recorre y llama a un método generar ficheroXSLX que sólo copia las filas de interés.

5) Utilizando la clase del punto1 o 2, probar las operaciones a nivel de fila y columna del DataFrame.

6) Partimos de 3 ficheros csv (pedidos, empresas y empleados). El fichero de pedidos tiene 2 columnas que contienen números que indican la empresa y el

empleado que gestiona el pedido. Queremos generar un csv de salida con esta estructura.

Tendremos que hacer merge entre los dataframe, borrar columnas, renombrar y reorganizar las columnas para conseguir el fichero resultante. Una vez que tengamos el DataFrame final exportarlo a un fichero csv, con sep=';'

Los 3 ficheros iniciales se encuentran en prácticas/Análisis de datos/merge

	idpedido	cliente	empresa	empleado	importe	pais
0	10248	WILMK	Federal Shipping	Buchanan	32.38	Finlandia
42	10249	TOMSP	Speedy Express	Suyama	11.61	Alemania
109	10250	HANAR	United Package	Peacock	65.83	Brasil
265	10251	VICTE	Speedy Express	Leverling	41.34	Francia
110	10252	SUPRD	United Package	Peacock	51.3	Belgica
266	10253	HANAR	United Package	Leverling	58.17	Brasil
1	10254	CHOPS	United Package	Buchanan	22.98	Suiza
392	10255	RICSU	Federal Shipping	Dodsworth	148.33	Suiza
267	10256	WELLI	United Package	Leverling	13.97	Brasil
111	10257	HILAA	Federal Shipping	Peacock	81.91	Venezuela

PARSEANDO XML:

1) A partir del fichero XML **libros.xml**, y utilizando el parseador basado en SAX. Generar un fichero CSV con los siguientes campos:

	A	B	C	D
1	anyo	titulo	editorial	precio
2	1994	TCP/IP Illustrated	Addison-Wesley	65.95
3	1992	Advan Programming for Unix environment	Addison-Wesley	65.95
4	2000	Data on the Web	Morgan Kaufmann editorials	39.95
5	1999	Economics of Technology for Digital TV	Kluwer Academic editorials	129.95

Para generar el fichero CSV, importar el módulo csv.

- import csv
- fich = open(fichero_csv, "w", newline='')
- writer = csv.writer(fich, delimiter=';')
- write.writerow(lista_de_cols)
- fich.close()

2) A partir de la **tabla de Empleados** en la base de datos **bbdd.dat de SQLite (en la caperta: practicas/xml)**, generar un fichero XML con los empleados.

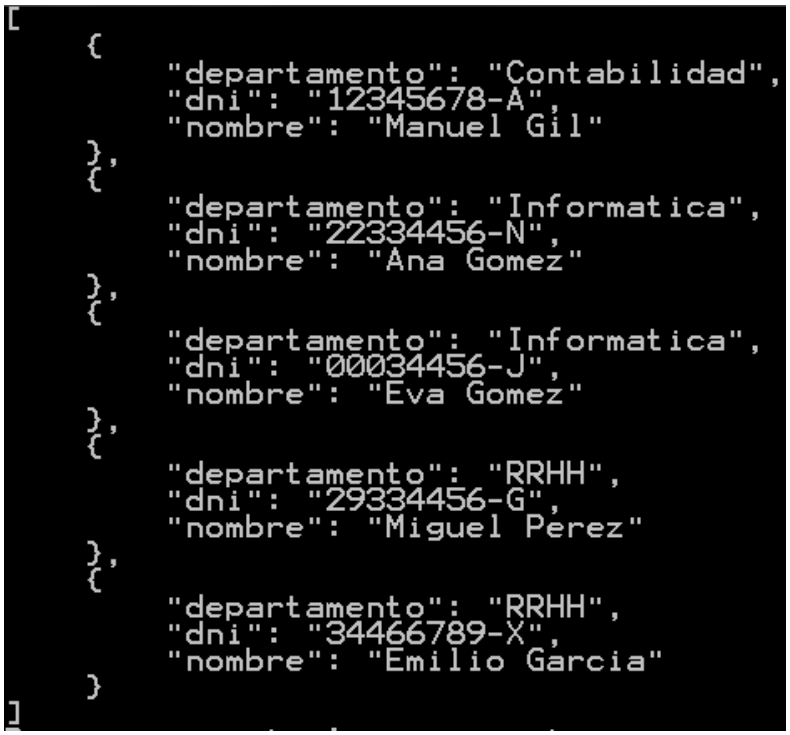
	dni	nombre	departamento
	Filter	Filter	Filter
1	12345678-A	Manuel Gil	Contabilidad
2	22334456-N	Ana Gomez	Informatica
3	00034456-J	Eva Gomez	Informatica
4	29334456-G	Miguel Perez	RRHH
5	34466789-X	Emilio Garcia	RRHH

El fichero generado:

```
<?xml version="1.0"?>
- <empleados>
  - <empleado dni="12345678-A">
      <nombre>Manuel Gil</nombre>
      <departamento>Contabilidad</departamento>
    </empleado>
  - <empleado dni="22334456-N">
      <nombre>Ana Gomez</nombre>
      <departamento>Informatica</departamento>
    </empleado>
  - <empleado dni="00034456-J">
      <nombre>Eva Gomez</nombre>
      <departamento>Informatica</departamento>
    </empleado>
  - <empleado dni="29334456-G">
      <nombre>Miguel Perez</nombre>
      <departamento>RRHH</departamento>
    </empleado>
  - <empleado dni="34466789-X">
      <nombre>Emilio Garcia</nombre>
      <departamento>RRHH</departamento>
    </empleado>
</empleados>
```

GENERAR JSON

1) Generar un fichero en Json con el contenido de los empleados de la BD.



**** nota: En la cabecera de los ficheros si añades acentos, … Codificación en UTF8**

 # -*- coding: utf-8 -*-