Estructuras de datos en Python

Anteriormente pudimos ver algo sobre los *Tipos de datos* que tiene Python (enteros, flotantes, cadenas, etc), pero ahora vamos hablar de tipos de datos más complejos que se constituyen en ***estructuras de datos***. Si pensamos en los primeros tipos de datos como ***átomos***, las estructuras de datos que vamos a ver serían ***moléculas***. Es decir, combinamos los tipos básicos en tipos más complejos.

Veremos distintas estructuras de datos como *listas, tuplas y diccionarios*.

➢ Listas

Las listas permiten almacenar *objetos* mediante un *orden definido* y con posibilidad de duplicados. Las listas son estructuras de datos *mutables*, lo que significa que podemos añadir, eliminar o modificar sus elementos.

Creando listas

Una lista está compuesta por cero o más elementos. En Python debemos escribir estos elementos separados por *comas* y dentro de *corchetes*. Veamos algunos ejemplos de listas:

```
Lista_vacia = []
languages = [‘Python’, ‘Ruby’, ‘Javascript’]
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13]
data = [‘Tenerife’, {‘cielo’: ‘limpio’, ‘temp’: 24}, 3718,
(28.2933947, -16.5226597)]
```

Nota: Una lista puede contener tipos de datos ***heterogéneos***, lo que la hace una estructura de datos muy versátil.

Conversión

Para convertir otros tipos de datos en una lista podemos usar la función ***list():***

```
# conversión desde una cadena de texto
list(‘Python’)
[‘P’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’]
```

Si nos fijamos en lo que ha pasado, al convertir la cadena de texto Python se ha creado una lista con 6 elementos, donde cada uno de ellos representa un caracter de la cadena.
Podemos extender este comportamiento a cualquier otro tipo de datos que permita ser iterado.

Operaciones con listas

Podemos obtener un elemento de una lista a través del **índice** (lugar) que ocupa. Ejemplo:

```
shopping = ['Agua', 'Huevos', 'Aceite']
print (shopping[0])
print (shopping[1])
```

```
print (shopping[2])
print (shopping[-1]) # acceso con índice negativo   ACEITE
print (shopping[-2]) # acceso con índice negativo   HUEVOS
print (shopping[-3]) # acceso con índice negativo AGUA
print (shopping[-4]) # ERROR DE ACCESO
print (shopping[4]) # ERROR DE ACCESO
```

El índice que usemos para acceder a los elementos de una lista tiene que estar comprendido entre los límites de la misma. Si usamos un índice antes del comienzo o después del final obtendremos un error.

***Nota**: Como se puede ver en las sentencias anteriores el índice de cada elemento puede ser positivo o negativo, si es positivo el primer elemento (de izquierda a derecha) tiene el índice 0 y crece hacia la derecha. Si se quiere acceder con índice negativo, la vinculación es en orden contrario, es decir, el último elemento tiene el valor -1, el penúltimo el -2 y así sucesivamente. Se recomienda que el acceso a los elementos se realice con índices positivos.*

Desglosar una lista

El desglose de una lista es con el objetivo de acceder a una parte de la lista principal, algo así como a una sublista de la misma:

```
shopping = [‘Agua’, ‘Huevos’, ‘Aceite’, ‘Sal’, ‘Limón’]
shopping[0:3]
[‘Agua’, ‘Huevos’, ‘Aceite’]
shopping[:3]
[‘Agua’, ‘Huevos’, ‘Aceite’]
shopping[2:4]
[‘Aceite’, ‘Sal’]
# Equivale a invertir la lista
shopping[::-1]
[‘Limón’, ‘Sal’, ‘Aceite’, ‘Huevos’, ‘Agua’]
```

Importante: Ninguna de las operaciones anteriores modifican la lista original, simplemente devuelven una lista nueva.

Invertir una lista

Python nos ofrece, al menos, tres mecanismos para invertir los elementos de una lista:

- ***Conservando la lista original***: Mediante troceado de listas con step negativo:

```
shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
shopping[::-1]
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

- ***Conservando la lista original:*** Mediante la función reversed():

```
shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
list(reversed(shopping))
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

- ***Modificando la lista original:*** Utilizando la función reverse() (nótese que es sin «d» al final):

```
shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
shopping.reverse()
shopping
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

Añadir al final de la lista

Una de las operaciones más utilizadas en listas es añadir elementos al final de las mismas. Para ello Python nos ofrece la función *append()*. Se trata de un método destructivo que modifica la lista original:

```
shopping = ['Agua', 'Huevos', 'Aceite']
shopping.append('Atún')
shopping
['Agua', 'Huevos', 'Aceite', 'Atún']
```

Creando desde vacío

Una forma muy habitual de trabajar con listas es empezar con una vacía e ir añadiendo elementos poco a poco. Se podría hablar de un **patrón creación**.
Supongamos un ejemplo en el que queremos construir una lista con los números pares del 1 al 20:

```
even_numbers = []
for i in range(20):
 if i % 2 == 0:
   even_numbers.append(i)

print(even_numbers)
```

El programa se encarga de crear una lista vacía y luego insertarle valores según la condición indicada. Se imprime lo siguiente:
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

Agregar en cualquier posición de una lista

Ya hemos visto cómo añadir elementos al final de una lista. Sin embargo, Python ofrece una función ***insert()*** que vendría a ser una generalización de la anterior, para incorporar elementos en cualquier posición. Simplemente debemos especificar el índice de inserción y el elemento en cuestión.

```
shopping = ['Agua', 'Huevos', 'Aceite']
shopping.insert(1, 'Jamón')
shopping
['Agua', 'Jamón', 'Huevos', 'Aceite']
shopping.insert(3, 'Queso')
shopping
['Agua', 'Jamón', 'Huevos', 'Queso', 'Aceite']
```

Nota: El índice que especificamos en la función ***insert()*** lo podemos interpretar como la posición delante (a la izquierda) de la cual vamos a colocar el nuevo valor en la lista.

Combinar listas

Python nos ofrece dos aproximaciones para combinar listas:

Conservando la lista original: Mediante el operador + o += (en este caso ninguna de las listas se modifica):

```
shopping = ['Agua', 'Huevos', 'Aceite']
fruitshop = ['Naranja', 'Manzana', 'Piña']
print (shopping + fruitshop)
```

Se imprime: ['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']

Modificando la lista original: Mediante la función extend(). Aquí la lista shopping se ve modificada:

```
shopping = ['Agua', 'Huevos', 'Aceite']
fruitshop = ['Naranja', 'Manzana', 'Piña']
shopping.extend(fruitshop)
print (shopping)
```

Se imprime: ['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']

Se podría pensar en el uso de ***append()*** para combinar listas. La realidad es que no funciona exactamente como esperamos; la segunda lista se añadiría como una sublista de la principal:

```
shopping = ['Agua', 'Huevos', 'Aceite']
```

```
fruitshop = ['Naranja', 'Manzana', 'Piña']
shopping.append(fruitshop)
print (shopping)
```

Se imprime: ['Agua', 'Huevos', 'Aceite', ['Naranja', 'Manzana', 'Piña']]

Otras formas de modificar listas

- ```
  shopping = ['Agua', 'Huevos', 'Aceite']
  shopping[0] = 'Leche'
  shopping[1] = 'Cafe'
  print (shopping)
  ```

- (con troceo, se reemplazan los elementos que ocupan la posición inicial y la posición final menos 1)
  ```
  shopping[1:3] = ['Atún', 'Pasta']
  print (shopping)
  ```

- Con la función **reverse()** se modifica la lista invirtiendo de posición sus valores:
  ```
  shopping = ['Agua', 'Huevos', 'Aceite']
  shopping.reverse()
  print (shopping)
  ```

  Imprime*: ['Aceite ', 'Huevos', 'Agua']*

- Creando una lista más personalizada y desde cero:

  ```
  numeros = []
  for i in range(20):
     if i % 2 == 0:
      numeros.append(i)

  print (numeros)
  ```

  Imprime*: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]*

- Añadir en cualquier posición de una lista
  ```
  shopping = ['Agua', 'Huevos', 'Aceite']
  shopping.insert(1, 'Jamón')
  print (shopping)
  ```

  Imprime: *['Agua', 'Jamón', 'Huevos', 'Aceite']*
  ```

Borrar elementos de una lista

Python nos ofrece, al menos, cuatro formas para borrar elementos en una lista:

Por su índice: Mediante la función ***del()***:

```
shopping = ['Agua', 'Huevos', 'Aceite']
del(shopping[1])
print (shopping)
```

Por su valor: Mediante la función ***remove()****(borra la primera aparición)*:

```
shopping = ['Agua', 'Huevos', 'Aceite']
shopping.remove('Huevos')
print (shopping)
```

Por su índice (con extracción): Las dos funciones anteriores del() y remove() efectivamente borran el elemento indicado de la lista, pero no «devuelven» nada. Sin embargo, Python nos ofrece la función ***pop()*** que además de borrar, nos «recupera» el elemento; algo así como una extracción. Lo podemos ver como una combinación de *acceso + borrado*:

```
shopping = ['Agua', 'Huevos', 'Aceite']
print (shopping.pop())
print (shopping)
```

Imprime: ['Agua', 'Huevos']

Por su rango: Mediante troceado de listas:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'chocolate', 'galletas']
shopping[1:3] = []
print (shopping)
```

Imprime: ['Agua', 'chocolate', 'galletas']

Borrado completo de la lista

Python nos ofrece, al menos, dos formas para borrar una lista por completo:

- Utilizando la función clear():

```
shopping = ['Agua', 'Huevos', 'Aceite', 'chocolate', 'galletas']
shopping.clear()
print (shopping)
```

- «Reinicializando» la lista a vacío con []:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'chocolate', 'galletas']
shopping = []
print (shopping)
```

Encontrar un elemento

Si queremos descubrir el índice que corresponde a un determinado valor dentro la lista podemos usar la función ***index()*** para ello (si el elemento esta más de una vez, nos devuelve el índice de la primera aparición):

```
shopping = ['Agua', 'Huevos', 'Aceite', 'chocolate', 'galletas']
print (shopping.index('Aceite'))
```

Imprime: 2

Longitud de una lista

Podemos conocer el número de elementos que tiene una lista con la función ***len():***

```
shopping = ['Agua', 'Huevos', 'Aceite', 'chocolate', 'galletas']
print (len(shopping))
```

Imprime: 5

Recorriendo una lista

Se puede recorrer una lista utilizando la estructura de control "***for***" y de esa manera acceder al contenido de la misma. Por ej.:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'chocolate', 'galletas']
for producto in shopping:
   print(producto)
```

Imprime: cada uno de los elementos de la lista, de manera individual.

Otro ejemplo, pero esta vez haciéndolo con su índice:

```
detalles = ['mineral natural', 'de oliva virgen', 'basmati']
for i in range(len(detalles)):
   print( detalles[i])
```

Este ejemplo imprime cada uno de los elementos de la lista, al igual que el ejemplo anterior.

Hay veces que no sólo nos interesa «visitar» cada uno de los elementos de una lista, sino que también queremos saber su índice dentro de la misma. Para ello Python nos ofrece la función ***enumerate()***:

```
shopping = ['Agua', 'Huevos', 'Aceite', 'chocolate', 'galletas']
for i, producto in enumerate(shopping):
   print(i, producto)
```

Imprime:
0 Agua

1 Huevos
2 Aceite
3 chocolate
4 galletas

Python ofrece la posibilidad de iterar sobre múltiples listas en paralelo utilizando la función ***zip()***:

```
shopping = ['Agua', 'Aceite', 'Arroz']
detalles = ['mineral natural', 'de oliva virgen', 'basmati']
for producto, detalle in zip(shopping, detalles):
   print(producto, " // " , detalle)
```

Imprime:

Agua // mineral natural
Aceite // de oliva virgen
Arroz // basmati

Nota: En el caso de que las listas no tengan la misma longitud, la función ***zip()*** realiza la combinación hasta que se agota la lista más corta.

Algunas Funciones matemáticas con listas en Python

Python nos ofrece, entre otras (si se quiere profundizar se puede utilizar la librería **Numpy**), las siguientes tres funciones matemáticas básicas que se pueden aplicar sobre listas.

- **Suma de todos los valores:** Mediante la función ***sum():***

```
datos = [5, 3, 2, 8, 9, 1]
print(sum(datos))
```

Imprime: 28

- **Mínimo de todos los valores:** Mediante la función ***min():***

```
datos = [5, 3, 2, 8, 9, 1]
print(min(datos))
```

Imprime: 1

- **Máximo de todos los valores:** Mediante la función ***max():***

```
datos = [5, 3, 2, 8, 9, 1]
print(max(datos))
```

Imprime: 9

Ejercicio para Practicar

Se solicita crear un programa que realice la carga de una lista con una cierta cantidad de nombres de productos comestibles, luego solicitarle al usuario que ingrese una fruta e indicarle si dicha fruta se encuentra en la lista precargada y en qué posición se encuentra.

➢ Tuplas

El concepto de ***tupla*** es muy similar al de lista. Aunque hay algunas diferencias menores, lo fundamental es que, mientras una lista es **mutable** y se puede modificar, una tupla no admite cambios y por lo tanto, es **inmutable.**

Podemos pensar en crear tuplas tal y como lo hacíamos con listas, pero usando paréntesis en lugar de corchetes:

```
tupla_vacia = ()
tupla_enteros =(5,10,20,30)
tupla_nombres = ('Juan','Pedro','Mario')
print(tupla_vacia)
print(tupla_enteros)
print(tupla_nombres)
```

Imprime:

```
()
(5, 10, 20, 30)
('Juan', 'Pedro', 'Mario')
```

Hay que prestar especial atención cuando vamos a crear una tupla de un ***único elemento***. La intención primera sería hacerlo de la siguiente manera:

```
tupla_vacia = ('fiesta')
print(tupla_vacia)
print("Tipo de Datos: ",type(tupla_vacia))
```

Imprime:

```
fiesta
Tipo de Datos:  <class 'str'>
```

Realmente, hemos creado una variable de tipo str (cadena de texto). Para crear una tupla de un elemento debemos añadir una coma al final:

```
tupla_vacia = ('fiesta',)
print(tupla_vacia)
print("Tipo de Datos: ",type(tupla_vacia))
```

Imprime:

```
('fiesta',)
Tipo de Datos:  <class 'tuple'>
```

Modificar una tupla

Como ya se dijo anteriormente, si se intenta modificar una tupla el interprete o herramienta de codificación nos indicara un error.

Conversión

Para convertir otros tipos de datos en una tupla podemos usar la función ***tuple()***:

```
nombres = ['Juan','Pedro','Mario']
print(tuple(nombres))
```

Esta conversión es válida para aquellos tipos de datos que sean iterables: cadenas de caracteres, listas, diccionarios, conjuntos, etc. Un ejemplo que no funciona es intentar convertir un número en una tupla:

```
tuple(5)
```

Se producirá un error.

Nivel avanzado (*usando la estructura match-case con tupla*)

La sentencia match-case va mucho más allá de una simple comparación de valores. Con ella podremos deconstruir estructuras de datos, capturar elementos o mapear valores. Para ejemplificar varias de sus funcionalidades, vamos a partir de una tupla que representará un punto en el plano (2 coordenadas) o en el espacio (3 coordenadas). Lo primero que vamos a hacer es detectar en qué dimensión se encuentra el punto:

```
point = (2, 5)
match point:
   case (x, y):
      print(f'({x},{y}) esta en el plano')
   case (x, y, z):
      print(f'({x},{y},{z}) esta en el espacio')
```

Se imprime por pantalla el mensaje . "(2,5) esta en el plano"

NOTA: el formato utilizado en el print es para facilitar la impresión de valores usando expresiones tipo string. Otras alternativas serianlas siguientes:

```
print("({},{}) esta en el plano".format(x, y))
ó
print("(" + str(x) + "," + str(y) + ") esta en el plano")


point = (2, 5, 10)
match point:
   case (x, y):
      print(f'({x},{y}) esta en el plano')
   case (x, y, z):
      print(f'({x},{y},{z}) esta en el espacio')
```

Se imprime por pantalla el mensaje . "(2,5,10) esta en el espacio"

Otra alternativa a lo anterior, es asegurarse de que los valores de la tupla sean enteros, para ello se deben convertir:

```
point = ('2', 5)
match point:
   case (int(), int()):
      print(f'({x},{y}) esta en el plano')
   case (int(), int(), int()):
      print(f'({x},{y},{z}) esta en el espacio')
   case _ :
      print('ERROR EN LOS DATOS')
```

Se imprime por pantalla el mensaje: "ERROR EN LOS DATOS", ya que uno de los datos NO es entero.

Operaciones con tuplas

Con las tuplas podemos realizar todas las operaciones que vimos con *listas* ***salvo las que conlleven una modificación*** «in-situ» de la misma:

- *reverse()*
- *append()*
- *extend()*
- *remove()*
- *clear()*
- *sort()*

Tuplas vs Listas

Aunque puedan parecer estructuras de datos muy similares, sabemos que las tuplas carecen de ciertas operaciones, especialmente las que tienen que ver con la modificación de sus

valores, ya que no son inmutables. Si las listas son más flexibles y potentes, ¿por qué íbamos a necesitar tuplas? Veamos 4 potenciales ventajas del uso de tuplas frente a las listas:

- Las **tuplas** ocupan menos espacio en memoria.
- En las **tuplas** existe protección frente a cambios indeseados.
- Las **tuplas** se pueden usar como **claves de diccionarios** (son «hashables»).
- Las **namedtuples** son una alternativa sencilla a los objetos.

Ejercicio para Practicar

Suponer una lista con datos de las compras hechas en un comercio de barrio, la cual contiene tuplas con información de cada venta: (cliente, producto, monto), por ejemplo: [('PEDRO', 'PAPA', 1200.0), ('JOSE', 'CAMOTE', 320.0), ('MARIA', 'PAPA', 600.0), ('ANDRES', 'CAMOTE', 500.0)].
El objetivo es obtener el monto total de todas la ventas realizadas para un producto en particular.

➢ Diccionarios

Podemos trasladar el concepto de diccionario de la vida real al de diccionario en Python.
Al fin y al cabo un diccionario es un objeto que contiene palabras, y cada palabra tiene asociado un significado. Haciendo el paralelismo, diríamos que en Python un diccionario es también un objeto indexado por **claves** (las palabras) que tienen asociados unos **valores** (los significados).

Los diccionarios en Python tienen las siguientes características:

- Mantienen el **orden** en el que se insertan las claves.
- Son **mutables**, con lo que admiten añadir, borrar y modificar sus elementos.
- Las **claves** deben ser **únicas**. A menudo se utilizan las cadenas de texto como claves, pero en realidad podría ser cualquier tipo de datos inmutable: enteros, flotantes, tuplas (entre otros).
- Tienen un **acceso muy rápido** a sus elementos, debido a la forma en la que están implementados internamente.

Nota: *En otros lenguajes de programación, a los diccionarios se los conoce como arrays*

Para crear un diccionario usamos llaves {} rodeando asignaciones ***clave: valor*** que están separadas por comas. Veamos algunos ejemplos de diccionarios:

```
diccionario_vacio = {}
diccionario_sueldo_años = {
   2019: 100000,
   2020: 110000,
   2021: 125000,
   2022: 140000
}
```

Operaciones con diccionarios

Para obtener un elemento de un diccionario basta con escribir la clave entre corchetes.

```
diccionario_sueldo_años = {
   2019: 100000,
   2020: 110000,
   2021: 125000,
   2022: 140000
}

print(diccionario_sueldo_años[2020])
```

Existe una función muy útil para «superar» los posibles errores de acceso por claves inexistentes. Se trata de ***get()*** y su comportamiento es el siguiente:

- Si la clave que buscamos existe, nos devuelve su valor.
- Si la clave que buscamos no existe, nos devuelve ***None***. No daría nunca error.

```
diccionario_sueldo_años = {
   2019: 100000,
   2020: 110000,
   2021: 125000,
   2022: 140000
}
print(diccionario_sueldo_años.get(2020)) #imprime el valor 110000
print(diccionario_sueldo_años.get(2026)) #imprime None
```

Añadir o modificar un elemento

Para añadir un elemento a un diccionario sólo es necesario hacer referencia a la clave y asignarle un valor:

- Si la clave ya existía en el diccionario, se reemplaza el valor existente por el nuevo.
- Si la clave es nueva, se añade al diccionario con su valor. No vamos a obtener un error a diferencia de las listas.

Partimos del siguiente diccionario para ejemplificar estas acciones:

```
diccionario_sueldo_años = {
   2019: 100000,
   2020: 110000,
   2021: 125000,
   2022: 140000
}
diccionario_sueldo_años[2018] = 95000
print(diccionario_sueldo_años.get(2020))
print(diccionario_sueldo_años.get(2018))
```

Creando desde cero

Una forma muy habitual de trabajar con diccionarios es utilizar el patrón creación partiendo de uno vacío e ir añadiendo elementos poco a poco.

```
diccionario_sueldo_años = {}
sueldo_inicial = 950000
for i in range(2018, 2022):
    sueldo_inicial += 50000
    diccionario_sueldo_años[i] = sueldo_inicial

print(diccionario_sueldo_años)
```

Otro ejemplo con más cosas:

```
diccionario_sueldo_años = {}
sueldo_inicial = 950000
for i in range(2018, 2022):
    sueldo_inicial += 50000
    diccionario_sueldo_años[i] = sueldo_inicial

for clave in diccionario_sueldo_años: #para imprimir las claves tb podriamos usar .keys()
    print(clave)

for sueldo in diccionario_sueldo_años.values():
    print(sueldo)
```

Iterar sobre “clave-valor”:

```
diccionario_sueldo_años = {}
sueldo_inicial = 950000
for i in range(2018, 2022):
    sueldo_inicial += 50000
    diccionario_sueldo_años[i] = sueldo_inicial

for clave, sueldo in diccionario_sueldo_años.items():
    print("Para la clave: ",clave, " corresponde el valor: ", sueldo)
```

Borrar elementos

Python nos ofrece, al menos, tres formas para borrar elementos en un diccionario:

- **Por su clave:** Mediante la sentencia del:

```
diccionario_sueldo_años = {
    2019: 100000,
    2020: 110000,
    2021: 125000,
    2022: 140000
}
```

```
del(diccionario_sueldo_años[2020])
```

- **Por su clave (con extracción):** Mediante la función **pop()** podemos extraer un elemento del diccionario por su clave. Vendría a ser una combinación de **get() + del**:

```
diccionario_sueldo_años = {
   2019: 100000,
   2020: 110000,
   2021: 125000,
   2022: 140000
}

diccionario_sueldo_años.pop(2020)
```

- **Borrado completo del diccionario:**
  - ✓ Utilizando la función clear():

```
diccionario_sueldo_años = {
   2019: 100000,
   2020: 110000,
   2021: 125000,
   2022: 140000
}
diccionario_sueldo_años.clear()
```

  - ✓ «Reinicializando» el diccionario a vacío con {}:

```
diccionario_sueldo_años = {
   2019: 100000,
   2020: 110000,
   2021: 125000,
   2022: 140000
}
diccionario_sueldo_años={}
```

Obtener las claves de un diccionario

- **Método: keys()**

```
diccionario_sueldo_años = {
   2019: 100000,
   2020: 110000,
   2021: 125000,
   2022: 140000
}
print(diccionario_sueldo_años.keys())
```

Obtener los valores de un diccionario

- **Método: values()**

```
diccionario_sueldo_años = {
   2019: 100000,
   2020: 110000,
   2021: 125000,
   2022: 140000
}
print(diccionario_sueldo_años. values())
```

Obtener la cantidad de elementos de un diccionario

- **Método: len()**

```
diccionario_sueldo_años = {
   2019: 100000,
   2020: 110000,
   2021: 125000,
   2022: 140000
}
print(len(diccionario_sueldo_años))
```

Python también cuenta con "***conjunto***" y las funciones necesarias para manejar "***archivos***", es super recomendable que cada uno de ustedes pueda profundizar sobre estos conceptos.

Ejercicio para Practicar

Se necesita llevar el registro de datos de los productos que tiene un comercio de barrio. Registrando su nombre y el stock actual. Realice un menú para ingresar datos, para obtener el listado de los productos que estan por debajo del stock minimo (10 unidades) y poder obtener todos los productos registrados