UD04 Estructuras de datos

Una estructura de datos es una forma particular de organizar información en un computador para que pueda ser utilizada de manera eficiente

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:

```
empty_list = []

languages = ['Python', 'Ruby', 'Javascript']

fibonacci = [0, 1, 1, 2, 3, 5, 8, 13]

data = ['Tenerife', {'cielo': 'limpio', 'temp': 24}, 3718,
(28.2933947, -16.5226597)]
```

Aviso: Aunque está permitido, NUNCA llames list a una variable porque destruirías la función que nos permite crear listas. Y tampoco uses nombres derivados como _list o list_ ya que no son nombres representativos que identifiquen el propósito de la variable.

Ejercicio: Cree una lista con las 5 ciudades que más le gusten.

Conversión

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

```
list('Python')
```

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 carácter de la cadena. Podemos extender este comportamiento a cualquier otro tipo de datos que permita ser iterado (iterables).

Otro ejemplo interesante de conversión puede ser la de los rangos. En este caso queremos obtener una lista explícita con los valores que constituyen el rango [0,9]:

```
list(range(10))
```

Lista vacía

Existe una manera particular de usar list() y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en una lista, con lo que obtendremos una lista vacía:

```
list()
```

Para crear una lista vacía, se suele recomendar el uso de [] frente a list(), no sólo por estar más normalizado sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

Operaciones con listas

Obtener un elemento

Igual que en el caso de las cadenas de texto, podemos obtener un elemento de una lista a través del índice (lugar) que ocupa. Veamos un ejemplo:

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

print(shopping[0])

print(shopping[1])

print(shopping[2])

print(shopping[-1])  # acceso con índice negativo
```

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 (excepción):

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

print(shopping[3])

print (shopping[-5])
```

Trocear una lista

El troceado de listas funciona de manera totalmente análoga al troceado de cadenas. Veamos algunos ejemplos:

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

print(shopping[0:3])

print(shopping[:3])

print(shopping[2:4])

print(shopping[-1:-4:-1])

# Equivale a invertir la lista
print(shopping[::-1])
```

En el troceado de listas, a diferencia de lo que ocurre al obtener elementos, no debemos preocuparnos por acceder a índices inválidos (fuera de rango) ya que Python los restringirá a los límites de la lista:

```
print(shopping[10:])

print(shopping[-100:2])

print(shopping[2:100])
```

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

Operaciones:

Invertir una lista

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

Conservando la lista original:

Opción 1: Mediante troceado de listas con step negativo:

```
print(shopping)

print(shopping[::-1])
```

Opción 2: Mediante la función reversed():

```
print(shopping)

print(list(reversed(shopping)))
```

Modificando la lista original:

Utilizando la función reverse() (nótese que es sin «d» al final):

```
print(shopping)

shopping.reverse()

print(shopping)
```

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')

print(shopping)
```

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 [0,20):

```
even_numbers = []

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

print(even_numbers)
```

Añadir 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. También se trata de una función destructiva

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

shopping.insert(1, 'Jamón')

print(shopping)

shopping.insert(3, 'Queso')

print(shopping)
```

Al igual que ocurría con el troceado de listas, en este tipo de inserciones no obtendremos un error si especificamos índices fuera de los límites de la lista. Estos se ajustarán al principio o al final en función del valor que indiquemos:

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

shopping.insert(100, 'Mermelada')

print(shopping)

shopping.insert(-100, 'Arroz')

print(shopping)
```

Aunque es posible utilizar insert() para añadir elementos al final de una lista, siempre se recomienda usar append() por su mayor legibilidad:

```
values = [1, 2, 3]
values.append(4)
print(values)

values = [1, 2, 3]
values.insert(len(values), 4)   # don't do it!
print(values)
```

Repetir elementos

Al igual que con las cadenas de texto, el operador * nos permite repetir los elementos de una lista:

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

print(shopping * 3)
```

Combinar listas

Python nos ofrece dos aproximaciones para combinar listas:

Conservando la lista original:

Mediante el operador + o +=:

```
shopping = ['Agua', 'Huevos', 'Aceite']
fruitshop = ['Naranja', 'Manzana', 'Piña']

print(shopping + fruitshop)
```

Modificando la lista original:

Mediante la función extend():

```
shopping = ['Agua', 'Huevos', 'Aceite']
fruitshop = ['Naranja', 'Manzana', 'Piña']

shopping.extend(fruitshop)

print(shopping)
```

Hay que tener en cuenta que extend() funciona adecuadamente si pasamos una lista como argumento. En otro caso, quizás los resultados no sean los esperados. Veamos un ejemplo:

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

shopping.extend('Limón')

print(shopping)
```

El motivo es que extend() «recorre» (o itera) sobre cada uno de los elementos del objeto en cuestión. En el caso anterior, al ser una cadena de texto, está formada por caracteres. De ahí el resultado que obtenemos.

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)
```

Modificar una lista

Del mismo modo que se accede a un elemento utilizando su índice, también podemos modificarlo:

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

print(shopping[0])

shopping[0] = 'Jugo'

print(shopping)
```

En el caso de acceder a un índice no válido de la lista, incluso para modificar, obtendremos un error:

```
shopping[100] = 'Chocolate'
```

MODIFICAR CON TROCEADO

No sólo es posible modificar un elemento de cada vez, sino que podemos asignar valores a trozos de una lista:

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

print(shopping[1:4])

shopping[1:4] = ['Atún', 'Pasta']

print(shopping)
```

Nota: La lista que asignamos no necesariamente debe tener la misma longitud que el trozo que sustituimos.

Borrar elementos

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

Por su índice:

Mediante la sentencia del:

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

del shopping[3]

print(shopping)
```

Por su valor:

Mediante la función remove():

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

shopping.remove('Sal')

print(shopping)
```

Nota: Si existen valores duplicados, la función remove() sólo borrará la primera ocurrencia.

Por su índice (con extracción):

La sentencia del y la función 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', 'Sal', 'Limón']

product = shopping.pop()  # shopping.pop(-1)
print(product)

print(shopping)

product = shopping.pop(2)
print(product)

print(shopping)
```

Nota: Si usamos la función pop() sin pasarle ningún argumento, por defecto usará el índice -1, es decir, el último elemento de la lista. Pero también podemos indicarle el índice del elemento a extraer.

Por su rango:

Mediante troceado de listas:

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

shopping[1:4] = []

print(shopping)
```

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', 'Sal', 'Limón']

shopping.clear()  # Borrado in-situ

print(shopping)
```

«Reinicializando» la lista a vacío con []:

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

shopping = []  # Nueva zona de memoria

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:

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

print(shopping.index('Huevos'))
```

Tener en cuenta que, si el elemento que buscamos no está en la lista, obtendremos un error:

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

print(shopping.index('Pollo'))
```

Nota: Si buscamos un valor que existe más de una vez en una lista, la función index() sólo nos devolverá el índice de la primera ocurrencia

Advertencia: En listas no disponemos de la función find() que sí estaba disponible para cadenas de texto.

Pertenencia de un elemento

Si queremos comprobar la existencia de un determinado elemento en una lista, podríamos buscar su índice, pero la forma normalizada de hacerlo es utilizar el operador in:

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

print('Aceite' in shopping)

print('Pollo' in shopping)
```

Nota: El operador in siempre devuelve un valor booleano, es decir, verdadero o falso.

Ejercicio:

Determine si una cadena de texto dada es un isograma, es decir, no se repite ninguna letra.

Ejemplos válidos de isogramas:

- lumberjacks
- background
- downstream
- six-year-old

Número de ocurrencias

Para contar cuántas veces aparece un determinado valor dentro de una lista podemos usar la función count():

```
sheldon_greeting = ['Penny', 'Penny', 'Penny']

print(sheldon_greeting.count('Howard'))

print(sheldon_greeting.count('Penny'))
```

Convertir lista a cadena de texto

Dada una lista, podemos convetirla a una cadena de texto, uniendo todos sus elementos mediante algún separador. Para ello hacemos uso de la función join() con la siguiente estructura:

Estructura de llamada a la función join():

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

print(','.join(shopping))
print(' '.join(shopping))
print('|'.join(shopping))
```

Hay que tener en cuenta que join() sólo funciona si todos sus elementos son cadenas de texto:

```
print(', '.join([1, 2, 3, 4, 5]))
```

Nota: Esta función join() es realmente la opuesta a la de split() para dividir una cadena.

Ejercicio:

Consiga la siguiente transformación:

12/31/20 → 31-12-2020

Ejercicio:

Dada una lista de números: 1, 25, 12, 4, 8 ,5 93, 4

Imprima la lista ordenada.

Ordenar una lista

Python proporciona, al menos, dos formas de ordenar los elementos de una lista:

Conservando lista original:

Mediante la función sorted() que devuelve una nueva lista ordenada:

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

print(sorted(shopping))
```

Modificando la lista original:

Mediante la función sort():

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

shopping.sort()

print(shopping)
```

Ambos métodos admiten un parámetro «booleano» reverse para indicar si queremos que la ordenación se haga en sentido inverso:

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

print(sorted(shopping, reverse=True))
```

Longitud de una lista

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

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

print(len(shopping))
```

Iterar sobre una lista

Al igual que hemos visto con las cadenas de texto, también podemos iterar sobre los elementos de una lista utilizando la sentencia for:

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

for product in shopping:
    print(product)
```

Nota: También es posible usar la sentencia break en este tipo de bucles para abortar su ejecución en algún momento que nos interese.

ITERAR USANDO ENUMERACIÓN¶

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():

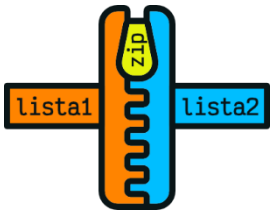
```
shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

for i, product in enumerate(shopping):
    print(i, product)
```

Truco: Es posible utilizar el parámetro start con enumerate() para indicar el índice en el que queremos comenzar. Por defecto es 0.

ITERAR SOBRE MÚLTIPLES LISTAS

Python ofrece la posibilidad de iterar sobre múltiples listas en paralelo utilizando la función zip(). Se basa en ir «juntando» ambas listas elemento a elemento:



Veamos un ejemplo en el que añadimos ciertos detalles a nuestra lista de la compra:

```
shopping = ['Agua', 'Aceite', 'Arroz']
details = ['mineral natural', 'de oliva virgen', 'basmati']

for product, detail in zip(shopping, details):
    print(product, detail)
```

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.

Dado que zip() produce un iterador, si queremos obtener una lista explícita con la combinación en paralelo de las listas, debemos construir dicha lista de la siguiente manera:

```
shopping = ['Agua', 'Aceite', 'Arroz']
details = ['mineral natural', 'de oliva virgen', 'basmati']

print(list(zip(shopping, details)))
```

Ejercicio:

Dados dos vectores (listas) de la misma dimensión, utilice la función zip() para calcular su producto escalar.

Entrada:

v1 = [4, 3, 8, 1]

v2 = [9, 2, 7, 3]

Comparar listas

Supongamos este ejemplo:

```
print([1, 2, 3] < [1, 2, 4])
```

Python llega a la conclusión de que la lista [1, 2, 3] es menor que [1, 2, 4] porque va comparando elemento a elemento:

El 1 es igual en ambas listas.

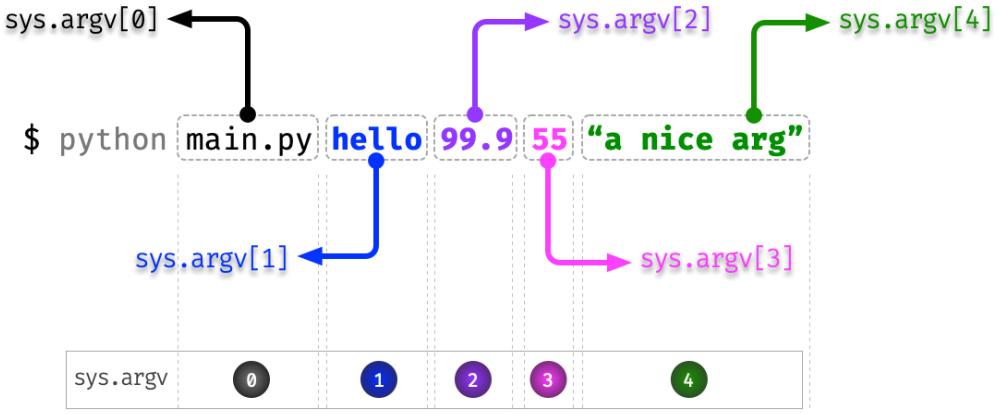
El 2 es igual en ambas litas.

El 3 es menor que el 4, por lo que la primera lista es menor que la segunda.

Entender la forma en la que se comparan dos listas es importante para poder aplicar otras funciones y obtener los resultados deseados.

sys.argv (Pendiente)

Cuando queramos ejecutar un programa Python desde línea de comandos, tendremos la posibilidad de acceder a los argumentos de dicho programa. Para ello se utiliza una lista que la encontramos dentro del módulo sys y que se denomina argv:




```
import sys

number = int(sys.argv[1])
tobase = int(sys.argv[2])

match tobase:
    case 2:
        result = f'{number:b}'
    case 8:
        result = f'{number:o}'
    case 16:
        result = f'{number:x}'
    case _:
        result = None

if result is None:
    print(f'Base {tobase} not implemented!')
else:
    print(result)
```

Funciones matemáticas

Python nos ofrece, entre otras 4, estas tres funciones matemáticas básicas que se pueden aplicar sobre listas.

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

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

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

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

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

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