Ficha 12

Arreglos Unidimensionales

1.] Introducción.

Hasta aquí se estudió la forma de resolver problemas de diversas características usando instrucciones y técnicas de programación variadas: algunos problemas requerían sólo del uso de condiciones y secuencias de instrucciones simples, en otros se hizo muy necesario el desarrollo de funciones y una fuerte división en subproblemas, y otros requerían ya el uso de ciclos.

Si observamos con detalle los últimos problemas que se han estado planteando (donde se utilizaron instrucciones repetitivas) y se los compara con los primeros problemas presentados en el curso (en los que los ciclos ni siquiera se conocían), podremos notar que las instrucciones de repetición se hicieron imperativamente necesarias para poder manejar un *gran volumen de datos* (o sea, una gran *cantidad* de datos). En los primeros problemas del curso, los datos que el programa necesitaba se cargaban *todos a la vez* en cierto conjunto pequeño de variables, y luego esas variables se procesaban usando instrucciones simples y condiciones [1].

Pero en los últimos problemas analizados, o bien era muy grande la cantidad de datos que pedía cada problema, o bien se desconocía en forma exacta la cantidad de datos que se presentarían. Ante esto, no resultaba ni práctico ni posible cargar todos los datos a la vez, dado que esto conduciría a definir un conjunto demasiado grande de variables que luego sería muy difícil de procesar. El uso de ciclos brindó una buena solución: se define un pequeño conjunto de variables necesario sólo para cargar *un dato o un subconjunto de datos*, se carga *un dato o un subconjunto*, se procesa el mismo, y se *repite el esquema* usando un ciclo, hasta terminar con todos los lotes de datos. El esquema que conocimos como *carga por doble lectura* es un ejemplo de esta técnica.

Si bien pudiera parecer que con esto quedan solucionados todos nuestros problemas, en realidad estamos muy lejos de tal situación. Sólo piense el lector lo que pasaría si el conjunto de datos tuviera que ser procesado *varias veces* en un programa. Con el esquema de "un ciclo procesando un dato a la vez", cada vez que el ciclo termina una vuelta se carga un *nuevo dato*, y se *pierde con ello el dato anterior*. Si como sugerimos, el programa requiriera volver a procesar un dato deberíamos volver a cargarlo, con la consecuente pérdida de tiempo y eficiencia general.

El re-procesamiento de un lote de datos es muy común en situaciones que piden *ordenar* datos y/o resultados de un programa, o en situaciones en las que un mismo conjunto de datos debe ser *consultado* varias veces a lo largo de una corrida (como el caso de los datos contenidos en una agenda, o los datos de la lista de productos de un comercio, por ejemplo).

Resulta claro que en casos como los planteados en el párrafo anterior, *no es suficiente* con usar un ciclo para cargar de a un dato por vez, perdiendo el dato anterior en cada repetición.

Ahora *se requiere que los datos estén todos juntos en la memoria*, y poder accederlos de alguna forma en el momento que se desee, sin perder ninguno de ellos.

¿Cómo mantener en memoria un volumen grande de datos, pero de forma que su posterior acceso y procesamiento sea relativamente sencillo? La respuesta a esta cuestión se encuentra en el uso de las llamadas *estructuras de datos*, que en forma muy general fueron presentadas en la *Ficha 03*.

Como vimos, esencialmente una *estructura de datos* es una variable que puede contener varios valores a la vez. A diferencia de las variables de *tipo simple* (que sólo contienen un único valor al mismo tiempo, y cualquier nuevo valor que se asigne provoca la pérdida del anterior), una *estructura de datos* puede pensarse como un conjunto de varios valores almacenados en una misma y única variable.

En ese sentido, sabemos que Python provee diversos tipos de *estructuras de datos* en forma de *secuencias*, y hemos usado con frecuencia algunas de ellas como las *cadenas de caracteres*, las *tuplas* y los *rangos*: en todos los casos, se trata de conjuntos de datos almacenados en una única variable [2].

Las *estructuras de datos* son útiles cuando se trata de desarrollar programas en los cuales se maneja un volumen elevado de datos y/o resultados, o bien cuando la organización de estos datos o resultados resulta compleja. Piense el alumno en lo complicado que resultaría plantear un programa que maneje los datos de todos los alumnos de una universidad, usando solamente variables simples como se hizo hasta ahora. Las *estructuras de datos* permiten agrupar en forma conveniente y ordenada todos los datos que un programa requiera, brindando además, muchas facilidades para acceder a cada uno de esos datos por separado.

Tan importantes resultan las estructuras de datos, que en muchos casos de problemas complejos resulta totalmente impracticable plantear un algoritmo para resolverlos sin utilizar alguna estructura de datos, o varias combinadas. No es casual entonces que la materia que estamos desarrollando se denomine *Algoritmos y Estructuras de Datos*...

2.] Arreglos unidimensionales en Python.

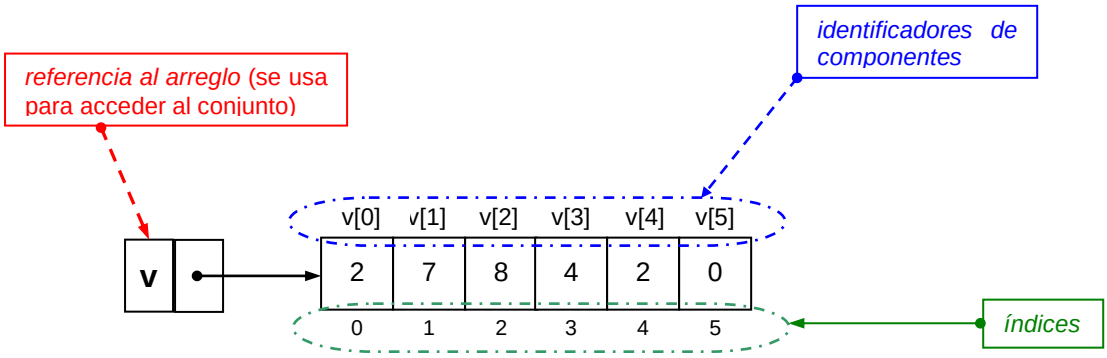
Una estructura de datos básica y muy importante en programación es la que se conoce como *arreglo unidimensional*. Se trata de una colección de valores que se organiza de tal forma que cada valor o componente individual es identificado automáticamente por un número designado como *índice*. El uso de los índices permite el acceso, uso y/o modificación de cada componente en forma individual.

La *cantidad de índices* que se requieren para acceder a un elemento individual, se llama *dimensión* del arreglo. Los *arreglos unidimensionales* se denominan así porque sólo requieren *un* índice para acceder a un componente [1]. Por otra parte, dada la similitud que existe entre el concepto de arreglo unidimensional y el concepto de vector en Álgebra, se suele llamar también *vectores* a los arreglos unidimensionales.

El gráfico de la *Figura 1* muestra la forma conceptual de entender un arreglo unidimensional. Se supone que la variable que permite manejar el arreglo se denomina *v* y que la misma contiene una referencia al arreglo (es decir, contiene la dirección de memoria donde está almacenado realmente el arreglo). En el ejemplo, el arreglo está dividido en seis casilleros,

de forma que en cada casillero puede guardarse un valor. Aquí supusimos que en cada casillero se almacenaron números enteros, pero obviamente podrían almacenarse valores de cualquier tipo (e incluso combinar valores de tipos distintos en un mismo arreglo).

Figura 1: Esquema básico de un arreglo unidimensional.



Observar que cada casillero[1] es automáticamente numerado con *índices*, los cuales en Python comienzan a partir del cero: la primera casilla del arreglo es subindicada con el valor cero, en forma automática. A partir de su índice cada elemento del arreglo referenciado por *v* puede accederse en forma individual usando el *identificador del componente*: se escribe el nombre de la variable que referencia al arreglo, luego un par de corchetes, y entre los corchetes el valor del índice de la casilla que se quiere acceder. En ese sentido, el *identificador del componente* cuyo índice es 2, resulta ser *v[2]*.

En Python, en general, se podría pensar que todos los tipos de *secuencias* provistas por el lenguaje (*cadenas*, *tuplas*, *rangos*, etc.) son esencialmente arreglos unidimensionales ya que para todas ellas sería aplicable la definición que acabamos de exponer. Sabemos que podemos acceder a un caracter de una cadena o a un elemento de una tupla a través de su índice y hemos aplicado esta idea en numerosas situaciones prácticas. Sin embargo, todos los tipos de secuencias que hemos visto y usado hasta ahora tenían una restricción: eran secuencias de tipo ***inmutable***, lo cual quería decir que una vez creada una cadena o una tupla (por ejemplo) *su contenido no podía cambiarse* y cualquier intento de modificar el valor contenido en alguno de los casilleros provocaría un error de intérprete [2] [3].

Por lo tanto, estrictamente hablando, las *secuencias de tipo inmutable provistas por Python no representan operativamente arreglos unidimensionales*, ya que el programador esperaría poder *modificar* el valor de un casillero en cualquier momento, sin restricciones. Y en ese sentido, Python provee entonces un tipo de secuencia designado como *lista* (o *list*) que es el tipo que finalmente permite a un programador gestionar un arreglo en forma completa.

[1] La idea de una variable compuesta por casilleros lleva a pensar en tableros de juegos, y desde esa idea surge la asociación con la película *Jumanji* del año 1995, dirigida por *Joe Johnston* y protagonizada por *Robin Williams* y *Kirsten Dunst* (que era apenas una adolescente en ese momento). Un par de niños encuentran un raro juego de tablero (basado en la idea de recorrer una selva) llamado *Jumanji* y cuando comienzan a jugar descubren que las situaciones imaginarias asociadas a cada casillero se convierten en realidad cuando un jugador cae en esa casilla, haciendo del mundo un caos… En el año 2005 se lanzó una película muy similar (considerada como una secuela de *Jumanji*) llamada *Zathura* (dirigida por *Jon Favreau* y protaginizada por *Josh Hutcherson* y *Kristen Stewart* antes de ser la famosa protagonista de la serie *Crepúsculo*). En esta versión, *Zathura* es también un juego de tablero que hace realidad lo que ocurre en cada casilla, pero basado en una aventura espacial.

Una variable de tipo *list* en Python representa una secuencia ***mutable*** de valores de cualquier tipo, de forma que cada uno de esos valores puede ser accedido y/o *modificado* a partir de su índice [2]. Es el tipo *list* el que permite crear y usar variables que representen arreglos unidimensionales operativamente completos en Python y por lo tanto, de aquí en adelante, siempre supondremos que al hablar de *arreglos* (de cualquier dimensión) estaremos refiriéndonos a variables de tipo *list* en Python.

Para crear un arreglo unidimensional *v* inicialmente vacío en Python, se debe definir una variable de tipo *list* en cualquiera de las dos formas que mostramos a continuación [2] [3]:

```
v = []
v = list()
```

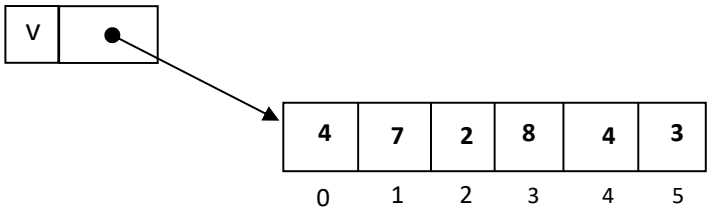
El par de corchetes vacíos representa justamente un arreglo vacío. La función constructora *list()* sin parámetros, también permite crear un arreglo vacío.

También se puede definir un arreglo no vacío por *enumeración de componentes*. La siguiente instrucción crea un arreglo *v* con seis componentes de tipo *int* (que en este caso serán los números 4, 7, 2, 8, 4 y 3):

```
v = [4, 7, 2, 8, 4, 3]
```

La instrucción anterior crea un arreglo de seis componentes, inicializa cada casilla de ese arreglo con los valores enumerados entre los corchetes, y retorna la dirección del arreglo (que en este caso se almacena en la referencia *v*):

Figura 2: Un arreglo con seis componentes de tipo int.



Observar que si el arreglo tiene 6(seis) elementos, entonces la última casilla del mismo lleva el índice 5 debido a que los índices comienzan desde el 0. *En un arreglo en Python, no existe una casilla cuyo índice coincida con el tamaño del arreglo.* Lo mismo vale para cualquier secuencia de cualquier tipo (cadenas, tuplas, etc.)

Al igual que con las cadenas y las tuplas, el contenido completo de una variable de tipo *list* puede mostrarse directamente en la consola estándar mediante una simple invocación a la función *print()*. La siguiente secuencia de instrucciones:

```
print('Contenido del arreglo:', end=' ')
print(v)
```

producirá la siguiente salida:

```
Contenido del arreglo: [4, 7, 2, 8, 4, 3]
```

Como queda dicho, un arreglo en Python puede contener valores de tipos distintos, aunque luego el programador deberá esforzarse por evitar un procesamiento incorrecto. La

instrucción siguiente crea un arreglo *datos* con cuatro componentes, que serán respectivamente una *cadena*, un *int*, un *boolean* y un *float*:

```
datos = ['Juan', 35, True, 5678.56]
```

Una vez que se creó el arreglo, se usa la referencia que lo apunta para acceder a sus componentes, ya sea para consultarlos o para modificarlos, colocando a la derecha de ella un par de corchetes y el índice del casillero que se quiere acceder. Los siguientes son ejemplos de las operaciones que pueden hacerse con los componentes de una variable de tipo *list* en Python, recordando que un variable de tipo *list* es *mutable* y por lo tanto los valores de sus casilleros pueden cambiar de valor (tomamos como modelo el arreglo *v* anteriormente creado con valores de tipo *int*):

```
# asigna el valor 4 en la casilla 3
v[3] = 4

# suma 1 al valor contenido en la casilla 1
v[1] += 1

# muestra en consola el valor de la casilla 0
print(v[0])

# asigna en la casilla 4 el valor de la casilla 1 menos el de la 0
v[4] = v[1] - v[0]

# carga por teclado un entero y lo asigna en la casilla 5
v[5] = int(input('Valor: '))

# resta 8 al valor de la casilla 2
v[2] = v[2] - 8
```

Si se necesita crear un arreglo (variable de tipo *list*) con cierta cantidad de elementos iguales a un valor dado (por ejemplo, un arreglo con *n* elementos valiendo cero), el truco consiste en usar el operador de *multiplicación* y repetir *n* veces una *list* que solo contenga a ese valor [2]:

```
ceros = 15 * [0]
print('Arreglo de 15 ceros:', ceros)
# muestra: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

La función constructora *list()* también puede usarse para crear una variable de tipo *list* a partir de otras secuencias ya creadas. Por ejemplo, el siguiente bloque de instrucciones:

```
x = 'Mundo'
letras = list(x)
print(letras)
# muestra: ['M', 'u', 'n', 'd', 'o']
```

crea primero una variable *x* como cadena de caracteres (que es un tipo particular de secuencia inmutable en Python). Luego se usa la función *list()* tomando a la cadena *x* como parámetro para crear un arreglo (una variable de tipo *list*) llamado *letras* que contendrá en cada casillero a cada una de las letras de la cadena *x* original. Lo mismo puede hacerse a partir de cualquier tipo de secuencia de datos de Python, como se ve en el siguiente bloque simple:

```
t = 3, 4, 5, 6
print('Tupla:', t)

v = list(t)
print('Lista:', v)
```

El ejemplo anterior crea primero una tupla *t* con cuatro números e inmediatamente muestra su contenido. Como se trata de una tupla, los valores aparecerán en consola encerrados entre *paréntesis*. Luego se crea una variable de tipo *list* llamada *v*, usando la función *list()* y pasándole como parámetro la tupla *t*. La lista o arreglo *v* se crea de forma que contiene los mismos valores que la tupla *t* y luego se muestra en consola. Como se trata de una lista los valores aparecerán encerrados entre *corchetes*:

```
Tupla: (3, 4, 5, 6)
Lista: [3, 4, 5, 6]
```

Si el arreglo se creó vacío y luego se desea agregar valores al mismo, debe usarse el método *append* provisto por el propio tipo *list* (que en realidad es una *clase*). Por ejemplo, la siguiente secuencia crea un arreglo *notas* inicialmente vacío, y luego se usa *append()* para agregar en él las cuatro notas de un estudiante [2] [3]:

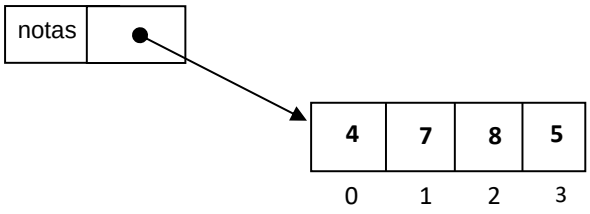
```
notas = []

notas.append(4)
notas.append(7)
notas.append(8)
notas.append(5)

print('Notas:', notas)
# muestra: Notas: [4, 7, 8, 5]
```

Note que el método *append()* agrega al contenido de la variable de tipo *list* el valor tomado como parámetro, pero lo hace de forma que ese valor se agrega *al final* del arreglo. En la secuencia anterior, el valor 4 quedará en la casilla 0 del arreglo, mientras que el 7, el 8 y el 5 quedarán en las casillas 1, 2 y 3 respectivamente, por orden de llegada:

Figura 3: Efecto de la función *append()* al crear una variable de tipo *list*.



Si se desea procesar un arreglo de forma que la misma operación se efectúe sobre cada uno de sus componentes, es normal usar un *ciclo for* de forma se aproveche la variable de control del ciclo como índice para entrar a cada casilla, realizando lo que se conoce como un *recorrido secuencial* del arreglo. El siguiente esquema muestra la forma de hacer la carga de cuatro valores por teclado para el mismo arreglo de notas que mostramos en el ejemplo anterior. El arreglo se muestra una vez cargado, y luego se recorre para obtener el promedio de las notas que contiene:

```
notas = []

for i in range(4):
    x = int(input('Nota: '))
    notas.append(x)

print('Notas:', notas)

suma = 0
for i in range(4):
    suma += notas[i]
promedio = suma / 4

print('Promedio:', promedio)
```

Recuerde que la invocación *range(4)* crea un rango o intervalo de cuatro elementos con los valores [0, 1, 2, 3] (comenzando en 0 y sin incluir al 4). Esto no es importante en el ciclo de carga del arreglo, pero sí lo es en el ciclo que acumula sus elementos para luego calcular el promedio, ya que el valor de la variable *i* se está usando como índice para entrar a cada casilla de la variable *notas*.

Siempre se puede usar la función *len()* para saber el tamaño o cantidad de elementos que tiene una variable de tipo *list*. La secuencia que calcula el promedio en el ejemplo anterior podría re-escribirse así:

```
suma = 0
n = len(notas)
for i in range(n):
    suma += notas[i]

promedio = suma / n
```

En Python también se puede definir una variable de tipo *list* empleando una técnica conocida como *definición por comprensión*, que dejaremos para ser analizada en la sección de *Temas Avanzados de esta Ficha*.

3.] Acceso a componentes individuales de una variable de tipo *list*.

El proyecto *[F12] Arreglos* dentro de la carpeta *Fuentes* que acompaña a esta Ficha, contiene algunos modelos desarrollados con *PyCharm* a modo de ejemplo de cada uno de los temas que analizaremos en esta sección.

Vimos que el tipo *list* se usa en Python para representar lo que en general se conoce como un arreglo en el campo de la programación. En muchos lenguajes de programación un arreglo es una estructura de *tamaño fijo*: esto significa que una vez creado, no se pueden agregar ni quitar casillas al arreglo, lo cual hace que los programadores deban seguir ciertas pautas para evitar un uso indebido de la memoria.

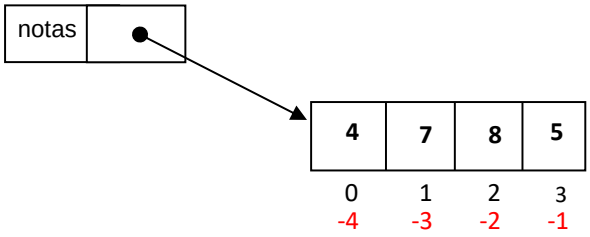
En ese sentido, un detalle a favor de Python es que *las variable de tipo list que se usan para representar arreglos no son de tamaño fijo*, sino que pueden aumentar o disminuir en forma dinámica su tamaño, agregar o quitar elementos en cualquier posición o incluso cambiar el tipo de sus componentes (como era de esperar en un lenguaje de tipado dinámico).

Al igual que para todos los tipos de secuencias, los elementos individuales de una variable de tipo *list* pueden accederse mediante índices. El primer elemento de una lista lleva el índice

cero, y los demás siguen en forma consecutiva. Pero además de esto, en Python se pueden usar *índices negativos* con cualquier tipo de secuencia (listas, tuplas, cadenas, etc.) sabiendo que el índice -1 corresponde al último elemento, el -2 al anteúltimo, y así sucesivamente [2].

Esto es: se puede asumir que en Python, cada casillero de cualquier tipo de *secuencia de datos* está identificado por dos números índices que pueden ser usados indistintamente por el programador, según lo crea conveniente: uno de esos índices es un número mayor o igual a 0 de forma que el 0 identifica a la primera casilla de la izquierda, y el otro es un número negativo menor o igual a -1 de forma que el -1 identifica *siempre* a la última casilla de la derecha. El arreglo *notas* que hemos mostrado como ejemplo más arriba, puede entenderse entonces como se ve en la gráfica siguiente:

Figura 4: Esquema de índices para una secuencia de cualquier tipo en Python.



El programa *test01.py* del citado proyecto *[F12] Arreglos* incluye una función general llamada *lists_test()* en la cual se crea una lista de números del 1 al *n* y luego se la recorre desde ambos extremos con ciclos *for* iterando sobre sus índices, para realizar algunas operaciones simples (*incluyendo un recorrido de derecha a izquierda usando índices negativos*):

```
# crear una lista con n numeros del 1 al n...
n = 10
numeros = []
for i in range(1, n+1):
    numeros.append(i)

print("Lista original: ", numeros)

# recorrer de izquierda a derecha y mostrar los numeros pares…
print("Valores pares contenidos en la lista: ", end=" ")
for i in range(len(numeros)):
    if numeros[i] % 2 == 0:
        print(numeros[i], end=" ")

# recorrer de derecha a izquierda y mostrar todos los numeros…
print("\nContenido de la lista, en forma invertida: ", end=" ")
for i in range(-1, (-len(numeros) - 1), -1):
    print(numeros[i], end=" ")
```

En Python, las variables de tipo *list* son de naturaleza *mutable*. Esto quiere decir que se puede cambiar el valor de un elemento sin tener que crear una *list* nueva. Lo siguiente, cambia el valor del elemento en la posición *2* por el valor *-1*:

```
numeros = [3, 5, 8, 6]
numeros[2] = -1
print(numeros)
# muestra: [3, 5, -1, 6]
```

Si se desea eliminar un elemento en particular, se puede recurrir a la instrucción *del*:

```
numeros = [3, 5, 8, 6, 9, 2]
del numeros[3]
print(numeros)  # muestra: [3, 5, 8, 9, 2]
```

Note que la instrucción *del* puede usarse para remover completamente toda la lista o arreglo, pero eso hace también que la variable que referenciaba al arreglo *quede indefinida de allí en más*. Una secuencia de instrucciones de la forma siguiente, provocará un error de variable no inicializada cuando se intente mostrar el arreglo:

```
numeros = [1, 2, 3]
del numeros
print(numeros) # error: variable no inicializada…
```

Todo tipo de secuencia en Python permite acceder a un subrango de casilleros *"rebanando" o "cortando" sus índices*. La siguiente instrucción crea una variable de tipo *list nueva* en la que se copian los elementos de los casilleros 1, 2 y 3 del arreglo original *numeros* [2]:

```
numeros = [4, 2, 3, 4, 7]
nueva = numeros[1:4]
print("\nNueva lista: ", nueva)
# muestra: [2, 3, 4]
```

La instrucción siguiente también usa un *corte de índices pero esta vez para copiar toda una lista*:

```
numeros = [2, 4, 6, 5]
copia = numeros[:] # copia toda la lista
print(copia) # muestra: [2, 4, 6, 5]
```

Llegados a este punto, es muy importante notar la diferencia entre asignar entre sí dos variables de tipo *list*, o usar *corte de índices* para copiar un arreglo completo. Si la variable *numeros* representa un arreglo, y se asigna en la variable *copia1*, entonces *ambas variables quedarán apuntando al mismo arreglo* (y por lo tanto, cualquier cambio en el contenido de uno afectará al contenido del otro):

```
numeros = [3, 5, 8, 3]

# una copia a nivel de referencias (copia superficial)...
copia1 = numeros  # ambas variables apuntan a la misma lista...

numeros[2] = -1
print("Original: ", numeros)
print("Copia 1: ", copia1)
```

El script anterior produce la siguiente salida:

```
Original: [3, 5, -1, 3]
Copia 1: [3, 5, -1, 3]
```

Sin embargo, cuando se usa el operador de *corte de índices* se está recuperando *una copia* de los elementos de la sublista pedida. Por lo tanto, el siguiente script ***no copia*** las referencias, sino que crea un segundo arreglo, cuyos elementos son iguales a los del primero (y cualquier modificación que se haga en una de las listas, no afectará a la otra):

```
numeros = [3, 5, 8, 3]

# una copia a nivel valores...
copia2 = numeros[:]  # cada variable es una lista diferente...

numeros[2] = -1
print("Original: ", numeros)
print("Copia 2: ", copia2)
```

El script anterior produce la siguiente salida:

```
Original: [3, 5, -1, 3]
Copia 2: [3, 5, 8, 3]
```

El acceso a subrangos de elementos mediante el corte de índices permite operaciones de consulta y/o modificación de sublistas en forma muy dinámica y flexible. En el programa *test01.py* se incluyen las siguientes secuencias de instrucciones que son en sí mismas muy ilustrativas. Los comentarios agregados en el código fuente ayudan a aclarar las ideas [2]:

```
# operaciones varias con cortes de indices...
lis = [47, 17, 14, 41, 37, 74, 48]
print('\nLista de prueba:', lis)

# reemplazar elementos en un rango...
lis[2:5] = [10, 20, 30]
print('Lista modificada 1:', lis)

# eliminar elementos en un rango...
lis[1:4] = []
print('Lista modificada 2:', lis)

# insertar un elemento... en este caso en la posicion 2...
lis[2:2] = [90]
print('Lista modificada 3:', lis)

# inserta una copia de si misma al principio...
lis[:0] = lis
print('Lista modificada 4:', lis)

# agregar un elemento al final, mediante un corte...
lis[len(lis):] = [100]
print('Lista modificada 5:', lis)

# agregar un elemento al final...
lis.append(200)
print('Lista modificada 6:', lis)

# copiar desde el casillero 2 hasta el ante-último...
l1 = lis[2:-1]
print('Sub-lista 1:', l1)

# copiar dos veces el subrango entre 2 y 4...
l2 = 2 * lis[2:5]
print('Sub-lista 2:', l2)

# concatenar dos listas...
l3 = lis[:4] + [1, 2, 3]
print('Sub-lista 3:', l3)
```

```
# vaciar una lista...
lis[:] = []
print('Lista original vacía:', lis)
```

En el modelo anterior se puede ver que usando cortes de índices es posible realizar casi cualquier operación de consulta o modificación. Observe que una instrucción como:

```
l2 = 2 * lis[2:5]
```

recupera desde el arreglo *lis* la sublista que incluye a los elementos desde la posición 2 hasta la posición 4, replica 2 veces esa sublista (debido al uso del operador *producto*) y asigna la sublista replicada en la variable *l2*.

Si se asigna *una lista* en un subrango de índices de otra lista, entonces la sublista que corresponde al subrango original será *reemplazada por la lista asignada*, aún si los tamaños no coinciden. Las instrucciones:

```
lis = [47, 17, 14, 41, 37, 74, 48]
lis[2:5] = [10, 20, 30]
print(lis)
# muestra: [47, 17, 10, 20, 30, 74, 48]
```

hace que los elementos que originalmente tenía el arreglo *lis* en los casilleros 2, 3 y 4 se reemplacen por los valores 10, 20 y 30 respectivamente. Si la lista asignada es de distinto tamaño que el subrango accedido, el reemplazo se lleva a cabo de todas formas, y la lista original puede ganar o perder elementos. En el siguiente ejemplo, un subrango de tres elementos es reemplazado por una lista de dos elementos (y el arreglo original queda con un elemento menos):

```
lis = [47, 17, 14, 41, 37, 74, 48]
lis[2:5] = [10, 20]
print(lis)
# muestra: [47, 17, 10, 20, 74, 48]
```

Y en el siguiente ejemplo, un corte de tres elementos es reemplazado por una lista de cuatro valores, y obviamente el arreglo original pasa a tener un elemento más:

```
lis = [47, 17, 14, 41, 37, 74, 48]
lis[2:5] = [10, 20, 30, 40]
print(lis)
# muestra: [47, 17, 10, 20, 30, 40, 74, 48]
```

Note también que si en un corte de índices el primer valor *i1* es mayor o igual al segundo *i2* (pero *i2* no es negativo), entonces el subrango accedido incluye solo *una lista vacía* en la posición *i1*. Por eso la instrucción:

```
lis[2:2] = [90]
```

en la práctica *agrega* el valor 90 exactamente en la posición 2 de la lista, pero no reemplaza al elemento en la posición 2 (en estricto rigor de verdad, *reemplaza la lista vacía* que "comienza en la posición 2 y termina en la 1", por la lista que contiene al 90). En ese sentido, sería exactamente lo mismo escribir

```
lis[2:1] = [90]    # o también lis[2:0] = [90]
```

Se deduce de lo anterior, que si quiere agregar un elemento (o una lista de varios elementos) *al final de la lista*, la siguiente instrucción permite hacerlo:

```
lis[len(lis):] = [100]
```

El corte usado en el ejemplo anterior comienza en la posición que coincide con el tamaño de la lista, y desde allí hasta el final. Como no existe una lista en esa posición (pero la posición en sí misma es válida) se considera que la sublista accedida es la *lista vacía* que se encuentra exactamente al final de la lista original. Y esa lista se reemplaza por la lista [100].

Es útil saber que en realidad lo que están haciendo las instrucciones vistas es asignar en el subrango pedido los elementos del arreglo que aparece en el lado derecho de la asignación, el cual es recorrido (iterado) de forma que cada componente se inserta en el arreglo de la izquierda. Sería un error asignar directamente un número, ya que en este caso no se tendría una secuencia iterable:

```
lis[2:2] = 90  # error!!!
```

Si el primer índice de un corte es positivo y el segundo es negativo, recuerde que el índice negativo *-1* accede al último elemento del arreglo, y por lo tanto, un corte de este tipo *no está reemplazando una lista vacía*, sino que reemplaza a un *subrango no vacío* perfectamente definido [2] [3]:

```
lis = [47, 17, 14, 41, 37, 74, 48]
lis[2:-1] = [10, 20]
print(lis)
# muestra: [47, 17, 10, 20, 48]
```

En el ejemplo anterior, el corte *lis[2 : -1]* está accediendo al subrango que comienza en el índice *2* (el *14* en el arreglo) y llega hasta el índice *-1* pero sin incluir a éste. Como el índice *-1* equivale al último casillero, entonces el corte llega hasta el anteúltimo (que en la lista sería el *74*). Y todos esos elementos (marcados en rojo en la lista original del ejemplo) se reemplazan por la lista *[10, 20]*.

4.] Aplicaciones y casos de análisis.

En esta sección se analizan y resuelven varios problemas para poner en práctica lo visto hasta ahora sobre *arreglos unidimensionales* [1].

Problema 34.) *Cargar por teclado un arreglo de n componentes y multiplicarlo por el valor k que también se ingresa por teclado.*

Discusión y solución: La lógica del programa a desarrollar es directa: En la función *test()* (que será la función de arranque del programa) primero se debe cargar por teclado el valor de *n*, que será usado para indicar cuántos casilleros tendrá el vector o arreglo (al hacer la carga se debe validar que el valor cargado no sea cero ni negativo). Después se puede crear un arreglo v que contenga ya n casillas inicializadas en 0 (con lo cual la carga del vector puede hacerse después accediendo a cada casilla por su índice, sin necesidad de recurrir a la función *append()* puesto que esas casillas ya existen en el arreglo)

Luego se puede desarrollar una función *read() que* cargue por teclado el arreglo, usando un ciclo *for* ajustado de forma que la variable de control *i* varíe desde 0 hasta *n – 1*. En cada

vuelta del ciclo se debe cargar el componente *v[i]* con lo que, al finalizar el ciclo, el arreglo quedará cargado completamente. Esa función está incluida en el módulo *arreglos.py* dentro del proyecto *[F12] Arreglos* y es la siguiente:

```
def read(v):
    # carga por teclado de un vector de n componentes...
    n = len(v)
    print('Cargue los elementos del vector:')
    for i in range(n):
        v[i] = int(input('casilla[' + str(i) + ']: '))
```

Finalmente podemos definir una función *product() que* también recorra el arreglo usando un *for*, pero en cada vuelta multiplique en forma acumulativa el componente *v[i]* por el valor *k*: el valor contenido en la casilla *v[i]* se multiplica por *k* y el resultado se vuelve a almacenar en la misma casilla *v[i]*. En el proyecto *[F12] Arreglos* que acompaña a esta Ficha, la función *product()* está incluida dentro del módulo *arreglos.py* y es la siguiente:

```
# función incluida en el módulo arreglos.py
def product(v, k):
    # multiplica por k al arreglo y lo retorna...
    n = len(v)
    for i in range(n):
        v[i] *= k
```

La solución a este problema se muestra a continuación (corresponde al modelo *testo02.py* del proyecto *[F12] Arreglos* que acompaña a esta ficha de clase):

```
__author__ = 'Cátedra de AED'

import arreglos


def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (>' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidio mayor a', inf, '... cargue de nuevo...')

    return n


def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear un arreglo de n elementos (valor inicial 0)...
    v = n * [0]

    # cargar el arreglo por teclado...
    arreglos.read(v)

    # cargar el número k por el cual se multiplicará al vector...
    k = int(input('Ingrese el valor de k: '))

    # multiplicar el vector por k...
    arreglos.product(v, k)

    # mostrar el vector final...
    print('El vector quedó así:', v)
```

```
# script principal...
if __name__ == '__main__':
    test()
```

Un ejemplo de pantalla de salida para este programa es el que sigue (suponiendo que se cargarán *n = 4* elementos en el vector y que el valor *k* será *k = 3*):

```
Ingrese cantidad de elementos (mayor a 0 por favor): 4
Cargue los elementos del vector:
casilla[0]: 3
casilla[1]: 4
casilla[2]: 5
casilla[3]: 6
Ingrese el valor de k: 3
El vector quedó así: [9, 12, 15, 18]
```

Veamos ahora otro problema sencillo:

Problema 35.) *Cargar por teclado dos vectores de tamaño n y obtener y mostrar el vector suma.*

Discusión y solución: La suma de dos vectores *a* y *b* de longitud *n* es otro vector *c* de longitud *n* en el cual cada componente es igual a la suma de las componentes con el mismo índice (u *homólogas*) en los vectores *a* y *b*. La función *add()* (que hemos incluido en el módulo *arreglos.py* del mismo proyecto *[F12] Arreglos*) efectúa la suma de *a* y *b* usando un ciclo *for*. En cada vuelta, toma el componente *a[i]*, lo suma con el componente *b[i]*, y guarda el resultado en el componente *c[i]*. Al finalizar el ciclo, el vector suma está completo y se lo devuelve mediante la instrucción *return*:

```
def add(a, b):
    # suma los arreglos a y b, y retorna el arreglo suma...
    n = len(a)
    c = n * [0]
    for i in range(n):
        c[i] = a[i] + b[i]

    return c
```

Aquí también usaremos la función *read()* del módulo *arreglos.py* que ahora será invocada dos veces para hacer la carga por teclado los arreglos *a* y *b*. Note cómo el uso de parámetros permite que una misma y única función pueda ser invocada tantas veces como sea necesario para procesar variables distintas. La función tiene un *parámetro formal* de tipo arreglo llamado *v*, el cual se equipara con cualquiera de los parámetros actuales (*a* o *b*) enviados desde *test()* y finalmente se hace la carga correcta. El programa completo es el siguiente, que corresponde al modelo *test03.py* del proyecto *[F12] Arreglos*:

```
__author__ = 'Cátedra de AED'

import arreglos


def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (> ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidio mayor a', inf, '... cargue de nuevo...')
```

```
    return n


def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear dos arreglos de n elementos (valor inicial 0)...
    a = n * [0]
    b = n * [0]

    # cargar los dos arreglos por teclado...
    print('\nVector a:')
    arreglos.read(a)
    print('\nVector b:')
    arreglos.read(b)

    # obtener el vector suma...
    c = arreglos.add(a, b)

    # mostrar el vector suma...
    print('\nEl vector suma es:', c)


# script principal...
if __name__ == '__main__':
    test()
```

Un ejemplo de ejecución y salida por pantalla de este programa se muestra a continuación:

```
Ingrese cantidad de elementos (mayor a 0 por favor): 3

Vector a:
Cargue los elementos del vector:
casilla[0]: 1
casilla[1]: 2
casilla[2]: 3

Vector b:
Cargue los elementos del vector:
casilla[0]: 4
casilla[1]: 5
casilla[2]: 6

El vector suma es: [5, 7, 9]
```

Analicemos ahora un último problema:

Problema 36.) *Cargar por teclado dos vectores de tamaño n y obtener el producto escalar de ambos.*

Discusión y solución: Recordemos que el *producto escalar (*o *producto interno* o también *producto punto)* entre dos vectores de longitud *n*, es un *número* (y ***no*** otro vector) que se calcula acumulando los productos de las componentes del mismo índice (u *homólogas*) de los dos vectores que se multiplican. El valor final de ese acumulador es el producto escalar entre los vectores.

En el programa que sigue, el producto escalar es calculado por la función *scalar_product()* incluida en el mismo módulo *arreglos.py* del proyecto *[F12] Arreglos*. Dicha función pone en cero el acumulador *sp*, y con un ciclo *for* efectúa los productos del tipo *a[i] * b[i]*. Cada uno

de esos productos se acumula en *sp*, y al terminar el ciclo se retorna el valor final contenido en ese acumulador:

```
# función incluida en el módulo arreglos.py
def scalar_product(a, b):
    # calcula el producto escalar entre a y b...
    n = len(a)

    sp = 0
    for i in range(n):
        sp += a[i]*b[i]

    return sp
```

Mostramos aquí el programa completo (ver modelo *test04.py* en el proyecto *[F12] Arreglos* que acompaña a esta Ficha):

```
__author__ = 'Cátedra de AED'

import arreglos


def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('cantidad de elementos (> ' + str(inf) + ' por favor): '))
        if n <= inf:
            print('Error: se pidio mayor a', inf, '... cargue de nuevo...')

    return n


def test():
    # cargar cantidad de elementos...
    n = validate(0)

    # crear dos arreglos de n elementos (valor inicial 0)...
    a = n * [0]
    b = n * [0]

    # cargar los dos arreglos por teclado...
    print('\nVector a:')
    arreglos.read(a)

    print('\nVector b:')
    arreglos.read(b)

    # obtener el producto escalar...
    pe = arreglos.scalar_product(a, b)

    # mostrar el prodcuto escalar...
    print('\nEl producto escalar es:', pe)


# script principal...
if __name__ == '__main__':
    test()
```

Anexo: Temas Avanzados

En general, cada Ficha de Estudios podrá incluir a modo de anexo un capítulo de *Temas Avanzados*, en el cual se tratarán temas nombrados en las secciones de la Ficha, pero que requerirían mucho tiempo para ser desarrollados en el tiempo regular de clase. El capítulo de *Temas Avanzados* podría incluir profundizaciones de elementos referidos a la programación en particular o a las ciencias de la computación en general, como así también explicaciones y demostraciones de fundamentos matemáticos. En general, se espera que el alumno sea capaz de leer, estudiar, dominar y aplicar estos temas aún cuando los mismos no sean específicamente tratados en clase.

a.) Definición *por comprensión* de variables de tipo *list*.

Hemos mostrado en la sección 2 de esta misma Ficha diversas formas de creación de variables de tipo *list* para representar arreglos unidimensionales en Python. Analizaremos ahora otra forma de creación de listas que consiste en *definir el contenido de la lista por comprensión*, recurriendo a alguna forma de iterador [2]. En el ejemplo siguiente (ver modelo *test05.py* del proyecto *[F12] Arreglos* que acompaña a esta Ficha), cada una de las letras de la cadena "*Hola*" es tomada por el ciclo iterador *for* y almacenada en la lista *letras* que se está creando:

```
# una cadena…
cadena = 'Hola'

# una lista creada por comprensión…
letras = [c for c in cadena]

print(letras)
# muestra: ['H', 'o', 'l', 'a']
```

Técnicamente, la variable *cadena* del ejemplo anterior puede ser de cualquier tipo *que admita un recorrido con iterador* (una cadena, una tupla, otra lista, etc.):

```
# una tupla…
secuencia = 1, 4, 7, 2

# una lista creada por comprensión a partir de la tupla…
numeros = [num for num in secuencia]

print(numeros)
# muestra: [1, 4, 7, 2]
```

El mismo arreglo *numeros* conteniendo los números del 1 al 10 que hemos creado en la sección 3 de esta Ficha mediante la siguiente secuencia de instrucciones:

```
# crear una lista con n numeros del 1 al n...
n = 10
numeros = []
for i in range(1, n+1):
    numeros.append(i)

print("Lista original: ", numeros)
```

podría equivalentemente ser creado mediante comprensión en la forma que mostramos a continuación:

```
# crear un arreglo con n numeros del 1 al n...
n = 10
numeros = [i for i in range(1, n+1)]
print('Lista original:', numeros)
```

La *creación de listas por comprensión* es una técnica sumamente flexible en Python. Sintácticamente, consiste en escribir dentro de un par de corchetes una expresión (que puede ser no solamente una variable…) seguida de un *for* que a su vez puede contener *otros ciclos for* y/o instrucciones condicionales *if*… El resultado será una nueva lista en la que cada uno de sus elementos será a su vez el resultado de cada una de las iteraciones del for sobre la expresión inicial.

Un ejemplo más complejo aclarará el panorama: supongamos que se desea crear una lista en la que cada elemento sea el cubo de los primeros 10 números naturales. Una forma de hacerlo (***sin*** aplicar *creación por comprensión*) sería:

```
cubos = []
for i in range(1, 11):
    cubos.append(i**3)
```

Pero aplicando *creación por comprensión*, lo anterior es exactamente lo mismo que:

```
cubos = [i**3 for i in range(1, 11)]
print('Lista de cubos:', cubos)
# muestra: [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Como sabemos, en el modelo anterior la expresión *i**3* equivale a elevar al cubo el valor de *i*, y a su vez el valor de *i* es aportado por el *for* en cada iteración sobre el rango de valores *[1..10]*.

Expresiones más complejas son también posibles. Supongamos que se tienen dos listas de números *v1* y *v2* y queremos generar una tercera lista que contenga tuplas o pares *(x, y)*, tales que *x* esté en *v1*, *y* esté en *v2* pero *x* sea menor que *y*. La forma básica de hacerlo ***sin*** aplicar comprensión sería:

```
v1 = [7, 3, 4, 8]
v2 = [1, 3, 5, 9]

pares = []
for x in v1:
    for y in v2:
        if x < y :
            pares.append((x, y))

print('Pares formados:', pares)
# muestra: [(7, 9), (3, 5), (3, 9), (4, 5), (4, 9), (8, 9)]
```

Pero *aplicando comprensión*, la creación de la lista se resume a:

```
v1 = [7, 3, 4, 8]
v2 = [1, 3, 5, 9]

pares = [(x, y) for x in v1 for y in v2 if x < y]
print('Pares formados:', pares)
# muestra: [(7, 9), (3, 5), (3, 9), (4, 5), (4, 9), (8, 9)]
```

Créditos

El contenido general de esta Ficha de Estudio fue desarrollado por el *Ing. Valerio Frittelli* para ser utilizada como material de consulta general en el cursado de la asignatura *Algoritmos y Estructuras de Datos* – Carrera de Ingeniería en Sistemas de Información – UTN Córdoba, en el ciclo lectivo 2019.

Actuaron como revisores (indicando posibles errores, sugerencias de agregados de contenidos y ejercicios, sugerencias de cambios de enfoque en alguna explicación, etc.) en general todos los profesores de la citada asignatura como miembros de la Cátedra, y particularmente la ing. Romina Teicher, que realizó aportes de contenidos, propuestas de ejercicios y sus soluciones, sugerencias de estilo de programación, y planteo de enunciados de problemas y actividades prácticas, entre otros elementos.

Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[2] Python Software Foundation, "Python Documentation," 2018. [Online]. Available: https://docs.python.org/3/.

[3] M. Pilgrim, "Dive Into Python - Python from novice to pro," 2004. [Online]. Available: http://www.diveintopython.net/toc/index.html.