Ficha 16

Arreglos Bidimensionales

1.] Introducción.

Hemos visto la manera de usar arreglos unidimensionales para procesar conjuntos grandes de datos sin tener que declarar un número elevado de variables de tipo simple. Hemos visto también la gran importancia que tiene el concepto de *acceso directo* a los componentes de un arreglo (de hecho, la característica principal de un arreglo es el acceso directo).

Se define como *dimensión de un arreglo* a la cantidad de índices que se requieren para acceder a uno de sus elementos. En ese sentido, los problemas que hemos analizado para resolver usando arreglos eran de naturaleza *unidimensional*: en todos los casos, los datos (o los resultados) se almacenaban en un arreglo de forma que para luego accederlos era suficiente conocer *un* índice (*y sólo uno*). En ningún caso enfrentamos la necesidad de organizar los datos de forma de accederlos con dos o más índices.

Sin embargo es muy común que esa necesidad se haga presente. Piense el estudiante en las siguientes situaciones cotidianas: un organizador de horarios (o simplemente un "horario") por lo general es una *tabla* que tiene una fila (horizontal) por cada día de la semana, y una columna (vertical) por cada bloque de horas que tenga sentido. En cada casilla de esa tabla de días y horas, normalmente se anota la actividad que debe desarrollarse en un día y hora particular. La consulta de esa tabla se hace entrando por la fila de un día dado (que actúa a modo de *primer índice*) y por la columna de una hora dada (que se usa como *segundo índice*). Como se ve, la organización de las actividades de una persona requiere, en este caso, considerar *dos índices* de acceso a la tabla la cual resulta entonces *bidimensional* (también se dice que la tabla es de *dos entradas*) [1].

No es el único ejemplo: una persona que quiera hacer un resumen de sus gastos en el año, suele plantear una tabla de dos entradas. En cada fila escribe uno de los rubros o ítems en los cuales efectuó algún gasto, en cada columna escribe el nombre de un mes del año, y finalmente en cada intersección de la tabla formada anota los importes que gastó en cada rubro en cada mes. Esta tabla tendrá doce columnas (una por cada mes), y tantas filas como rubros quiera controlar la persona. Se suele llamar *orden de una tabla* al *producto expresado* de la cantidad de filas por la cantidad de columnas. Si los rubros a controlar fueran diez, entonces la tabla del ejemplo sería de *orden 10*12* (observar que no importa tanto el resultado del producto, sino sólo dejarlo expresado).

La forma de implementar el concepto de *tabla bidimensional* o de *dos entradas* es usar *arreglos bidimensionales* (también llamados comúnmente *matrices*) y en Python esto implica la idea de *listas de listas* (variables de tipo *list* que en cada casilla contienen a otra *list*) [2]. Como veremos a lo largo de esta Ficha, la definición y uso de un arreglo de este tipo es una extensión natural del concepto de *arreglo unidimensional* ya estudiado.

2.] Creación y uso de arreglos bidimensionales en Python.

Básicamente, un *arreglo bidimensional* o *matriz* es un arreglo cuyos elementos están dispuestos en forma de tabla, con varias filas y columnas. Aquí llamamos *filas* a las disposiciones horizontales del arreglo, y *columnas* a las disposiciones verticales.

Hemos visto que en Python un arreglo unidimensional se implementa como una variable de tipo *list*. Pero sabemos también una colección de tipo *list* puede contener otras *list* embebidas en ella, y este hecho es el que permite definir arreglos bidimensionales en Python. La forma más directa y simple de hacerlo es la *asignación directa de valores constantes*: Está claro que si sabemos de antemano qué valores necesitamos en cada *fila* y *columna*, podemos asignar en forma directa la lista de listas que finalmente será nuestra matriz [2]:

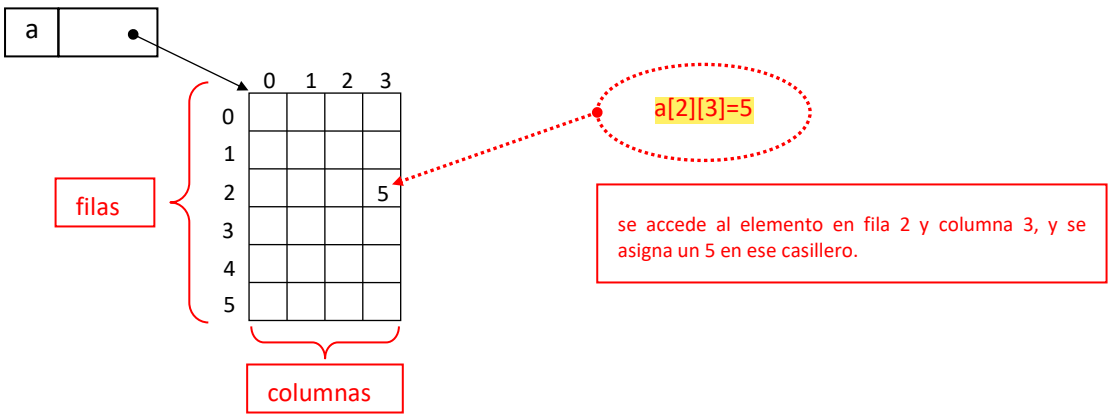
```
m0 = [ [1, 3, 4],
       [3, 5, 2],
       [4, 7, 1]
] # se puede indentar aqui o en la columna del corchete de apertura...

print('Matriz con valores fijos:', m0)
```

En el ejemplo anterior, la *lista de listas* se escribió en renglones separados e indentada, por razones de claridad, pero es totalmente equivalente a hacerlo en una sola línea:

```
m0 = [ [1, 3, 4], [3, 5, 2], [4, 7, 1] ]
print('Matriz con valores fijos:', m0)
```

Si la matriz está ya creada, para entrar a un componente debe darse el índice de la *fila* del mismo y también el índice de la *columna*. Como los índices requeridos son dos, el arreglo entonces es de dimensión dos. El siguiente esquema ilustra la manera de declarar y crear un arreglo bidimensional de orden 6*4 (conteniendo números enteros) en Python, la forma conceptual de representarlo, y la manera de acceder a uno de sus componentes [1]:

Figura 1: Esquema de representación de una matriz de orden 6*4.




```
a = [ [2, 3, 4, 5],
      [1, 5, 6, 3],
      [7, 2, 4, 0],
      [8, 4, 1, 7],
      [0, 0, 2, 9],
      [1, 3, 4, 1],
    ]

a[2][3] = 5  # accede a una casilla y cambia su valor.
```

En el ejemplo anterior, suponemos que casillero de la matriz[1] queda valiendo los valores asignados, y luego específicamente el casillero *a[2][3]* cambia su valor inicial (que era un 0) por un 5.

Para lo que sigue, el proyecto *F[16] Arreglos Bidimensionales* que acompaña a esta Ficha contiene un primer modelo llamado *test01.py* en el cual se implementan todas las técnicas que mostraremos a continuación.

Si en lugar de una matriz con valores iniciales fijos queremos crear en Python una lista de listas que represente una *matriz de n filas y m columnas*, con *n* y *m* variables, el inconveniente es que Python es un lenguaje de tipado dinámico y no es posible indicarle al intérprete en forma directa y simple cuántos elementos queremos que reserve para nuestra matriz en cada dimensión. Eso lleva a que las *matrices* deban ser construidas de alguna forma en tiempo de ejecución [2] [3].

Hay varias formas de hacer esto. Comencemos por la más descriptiva (pero también la más larga y posiblemente la más ineficiente). Supongamos que queremos *n* filas (suponga *n = 3*) y *m* columnas (suponga *m = 4*). La idea es partir de una lista inicialmente vacía, y asignar en ella *n* listas vacías:

```
m1 = []
for f in range(n):
    m1.append([])
```

El aspecto que tendría la matriz *m1* en este momento es algo como:

```
m1 ⇨ [[], [], []]
```

Cada una de las *n = 3* listas vacías de *m1* puede entenderse como una de las filas de la matriz que estamos construyendo, y claramente se acceden como *m1[0]* la primera, *m1[1]* la segunda, y *m1[2]* la tercera.

El paso final, es agregar en cada una de las *n* listas *m1[f]* un total de *m* valores del tipo que se requiera, lo cual ciertamente "abrirá" espacio en cada fila generando las columnas buscadas. Si no estamos seguros de qué tipo de valores necesitaremos almacenar, podemos usar *None*:

```
m1 = []
for f in range(n):
    m1.append([])
    for c in range(m):
        m1[f].append(None)
```

[1] En este contexto una *matriz* es una colección de datos organizados en forma de tabla… pero en la famosísima película *Matrix* (de 1999) esa *matrix* (o *matriz*) es un mundo virtual al cual la humanidad está conectada sin saberlo. Las computadoras tomaron el control del mundo y vencieron a los seres humanos en una devastadora guerra, y ahora los tienen prisioneros en la gran simulación que es la *Matrix*, a la cual no sólo están sometidos sino que también alimentan con la energía de sus cuerpos. Un puñado de humanos libres lucha contra la *Matrix*, mientras espera la llegada de un mesías que cumplirá la profecía de derrotar a las máquinas y liberar a la especie humana (con lo que el argumento cae en un fastidioso lugar común…) La película fue dirigida por los *hermanos Wachowsky* en 1999 (hoy esos hermanos cambiaron de sexo y se conocen como las *hermanas Wachowsky*) y fue protagonizada por *Keanu Reeves*, *Laurence Fishburne* y *Carrie-Ann Moss*. Hubo dos conocidas secuelas: *The Matrix Reloaded* (de 2003) y *The Matrix Revolutions* (también de 2003) para conformar así una trilogía que ya se ha convertido en objeto de culto para sus seguidores.

La matriz así construida tendría el siguiente aspecto final, con *n = 3* y *m = 4*:

```
m1 ⇨ [[None, None, None, None], [None, None, None, None], [None, None, None, None]]
```

O bien:

```
m1 ⇨  [
        [None, None, None, None],
        [None, None, None, None],
        [None, None, None, None]
      ]
```

Si en lugar de *n*m* valores *None* se quisiera disponer de *n*m* valores *0(cero)*, lo único que debe hacerse es reemplazar el valor *None* del ejemplo anterior por el valor *0*.

Otra forma de crear la matriz, no tan detallada ni tan intuitiva pero más compacta y eficiente, consiste en comenzar creando las *n* filas con el operador de multiplicación, de forma que arranquen con valores *None* (o cero o lo que se requiera):

```
m2 = [None] * n    # crea n componentes None (serán las filas...)
```

Si *n = 3*, el aspecto hasta aquí sería el siguiente, con los elementos *None* ya creados y por lo tanto con el espacio (y el índice…) ya asociado a ellos:

```
m2 ⇨ [None, None, None]
```

Para completar la matriz, se itera sobre los *n* elementos ya creados, y se los reemplaza por una lista (creada con el operador multiplicación) de *m* elementos *None* o del tipo que se prefiera. Como Python es de tipado dinámico, los valores *None* originalmente asignados en las *n* posiciones cambiarán a lo que sea que le indique [2] [3]:

```
m2 = [None] * n
for f in range(n):
    m2[f] = [None] * m  # ...expande cada fila a 4 elementos None
```

El resultado es, otra vez, una matriz de *n*m* elementos *None*. Si en lugar de valores *None* se quisieran valores *0(cero)* o cualesquiera otros, solo se debe reemplazar el segundo *None* por *0* o por el valor que se quiera. En rigor, el primer *None* carece de importancia y podría en la práctica ser cualquier valor simple, ya que durante la corrida del ciclo *for* esos *None* serán reemplazados por las listas de tamaño *m*.

Finalmente, una tercera vía para crear una matriz de *n*m* elementos consiste en usar alguna *expresión de comprensión*, lo cual es aún más compacto y eficiente, aunque posiblemente menos claro. Si analizamos con atención el modelo de creación que acabamos de mostrar, notaremos que el ciclo *for* realiza una iteración de *n* giros, y en cada uno de esos giros crea una lista de *m* elementos *None*. En rigor, eso es todo lo que se necesita para crear la matriz, si la expresión comprensiva se encierra entre corchetes [2]:

```
m3 = [[None] * m for f in range(n)]
```

Note que en este caso no se requiere indicar en qué casillero se asigna la lista de tamaño *m* que se está creando: solo se crean *n* listas de tamaño *m*, y cada una de ellas *se inserta en la lista representada por los corchetes externos*. Cada una llevará un índice desde *0* en adelante, por orden de llegada. El resultado es el esperado: una matriz de *n * m* elementos *None*. Reemplace el *None* por el valor que necesite, si fuese el caso.

Como vimos, una vez que la matriz ha sido creada el acceso a sus elementos individuales se hace con dos índices: el primero selecciona la "fila" (o sea, una de las n sublistas o subarreglos) y el segundo selecciona la "columna" (o sea, el elemento dentro de la sublista que fue seleccionada con el primer índice):

```
m1[0][3] = 10
m1[1][2] = 20
```

Observar que para acceder a un elemento, se escribe primero el nombre de la referencia al arreglo (*m* en el caso del ejemplo), luego el número de la *fila* del elemento que se quiere acceder, pero encerrado entre corchetes, y por último el número de la *columna* de ese elemento, también encerrado entre corchetes. Notar además, que en el lenguaje Python los arreglos de cualquier dimensión están *referidos a cero*, lo cual significa que el *primer índice de cada dimensión es siempre cero*. En la *Figura 1* (*página 322*) puede verse que el arreglo tiene seis filas, pero numeradas del 0(cero) al 5(cinco), y cuatro columnas, numeradas del 0(cero) al 3(tres). No hay excepciones a esta regla, por lo cual debe tenerse cuidado de ajustar correctamente los ciclos para el recorrido de índices.

Si se quiere *recorrer por filas* en forma completa una matriz de *n*m* elementos, la idea es prácticamente igual a la forma de hacerlo en cualquier otro lenguaje. El siguiente ejemplo asigna en cada elemento el producto entre su número de fila y su número de columna:

```
# un recorrido por filas
for f in range(len(m2)):
    for c in range(len(m2[f])):
        m2[f][c] = f * c
```

El primer ciclo *for* recorre con la variable *f* el rango de índices de las "filas" (o sublistas) de la matriz. Note que la función *len()* aplicada sobre la variable *m2* que representa a la matriz completa, retornará la cantidad de sublistas (filas) que *m2* contiene. El segundo ciclo *for* recorre con la variable *c* el rango de índices de la sublista *m2[f]* (o sea, las "columnas" de esa fila). De nuevo, note que la expresión *len(m2[f])* retorna la cantidad de elementos de la sublista *f* (la cantidad de columnas de la fila *f*). De hecho, con este planteo se podría recorrer sin problemas una matriz "dentada", cuyas filas tuviesen tamaños diferentes [1].

Ligeramente diferente es el problema si se quiere *recorrer la matriz por columnas*. Recuerde que en esencia, lo que tenemos es una lista de listas en la que las sublistas (primera dimensión) representan las filas, pero en la segunda dimensión *no tenemos otras listas* sino directamente los elementos a acceder. Para el recorrido por columnas, se deben invertir los dos ciclos *for*: llevar más afuera el que recorre las columnas (*for c*) y más adentro el que recorre la filas (*for f*). El tema es que ahora no podemos usar la función *len()* en el ciclo externo, ya que la fila a la que corresponde el índice *c* aún no ha sido seleccionada. En principio, debemos asumir que la matriz es regular (no dentada) y conocer de antemano la cantidad de columnas:

```
# recorrido por columnas – matriz regular
filas = 3
columnas = 4
for c in range(columnas):
    for f in range(filas):
        m3[f][c] = f * c
```

El recorrido completo de una matriz (por filas o por columnas) en definitiva equivale al *recorrido secuencial de esa matriz* y es un proceso muy común: el programador deberá aplicarlo cada vez que desee procesar todos y cada uno de los elementos de una matriz. Por ejemplo, la carga por teclado de una matriz *m4* de 3 filas y 4 columnas se puede hacer mediante un recorrido del tipo que prefiera el programador. En este caso, aplicamos el *recorrido por filas* que ya hemos citado: dos ciclos *for* anidados, de forma que el primero recorra las filas de la matriz, y el segundo las columnas [1]:

```
# carga por teclado... recorrido por filas en orden creciente...
filas, columnas = 3, 4
m4 = [[0] * columnas for f in range(filas)]
for f in range(filas):
    for c in range(columnas):
        m4[f][c] = int(input('Valor: '))
print('Matriz 4 leida:', m4)
```

Otra vez, la idea básica del proceso es que la variable *f* del ciclo más externo se usa para indicar qué *fila* se está procesando en cada vuelta. Dado un valor de *f*, se dispara otro ciclo controlado por *c*, cuyo objetivo es el de recorrer todas las *columnas* de la fila indicada por *f*. Notar que mientras avanza el ciclo controlado por *c* permanece fijo el valor de *f*. Sólo cuando corta el ciclo controlado por *c*, se retorna al ciclo controlado por *f*, cambiando ésta de valor y comenzando por ello con una nueva fila. El proceso de recorrer secuencialmente una matriz avanzando fila por fila empezando desde la cero, como aquí se describe, se denomina *recorrido en orden de fila creciente*.

Como vimos, el mismo proceso de carga (o el que sea que requiera el programador) se puede hacer con recorridos de otros tipos, simplemente cambiando el orden de los ciclos. El siguiente esquema realiza un *recorrido en orden de fila decreciente*: comienza con la última fila, y barre cada fila hacia atrás hasta llegar a la fila cero [1]:

```
# carga por teclado... recorrido por filas en orden decreciente...
filas, columnas = 3, 4
m5 = [[0] * columnas for f in range(filas)]
for f in range(filas-1, -1, -1):
    for c in range(columnas):
        m5[f][c] = int(input('Valor: '))
print('Matriz 5 leida:', m5)
```

Notar que el cambio sólo consistió en hacer que la variable *f* (usada para barrer las filas), comience en el valor *filas–1* (que es el índice de la última fila), y se *decremente* hasta llegar a cero. El ciclo controlado por *c* se dejó como estaba.

Si se desea un *recorrido en orden de columna creciente* (*o decreciente*), sólo deben invertirse los ciclos: el ciclo que *recorre las columnas* (controlado por *c* en nuestro ejemplo) debe ir por fuera, y el ciclo que recorre las filas (controlado por *f* en este caso) debe ir por dentro. De esta forma, el valor de *c* no cambia hasta que el ciclo controlado por *f* termine todo su recorrido. Sin embargo, no debe olvidarse que si queremos que *c* indique una columna, entonces *c* debe usarse en el ***segundo par de corchetes*** al acceder a la matriz. Y si la variable *f* va a indicar filas, entonces debe usarse en el ***primer par de corchetes***. Esto es independiente del orden en que se presenten los ciclos para hacer cada recorrido [1]:

La variable que indica la *fila* va en el *primer par de corchetes*, y la variable que indica la *columna* va en el *segundo par de corchetes*, sin importar cuál ciclo va por fuera y cuál por dentro.

El siguiente esquema muestra un recorrido en *orden de columna creciente*, para leer por teclado una matriz *m6* de *n* filas y *m* columnas:

```
# carga por teclado... por columnas en orden creciente...
filas, columnas = 3, 4
m6 = [[0] * columnas for f in range(filas)]
for c in range(columnas):
    for f in range(filas):
        m6[f][c] = int(input('Valor: '))
print('Matriz 6 leida:', m6)
```

3.] Totalización por filas y columnas.

En muchas situaciones se requerirá procesar los datos contenidos en una matriz, de forma de obtener los totales acumulados de cada una de sus filas y/o de cada una de sus columnas. Los procesos para realizar estas tareas son sencillos, y se deducen directamente de los procesos de recorrido por filas y por columnas que ya hemos analizado. Veamos un problema típico a modo de caso de análisis [1]:

Problema 43.) *Un comercio mayorista trabaja con cierta cantidad **n** de artículos, numerados del 0 al **n-1**. Dispone de un plantel de **m** vendedores para su venta, los cuales están enumerados del 0 al **m-1** inclusive, en forma contigua. El gerente de dicho comercio desea obtener cierta información estadística respecto de las ventas realizadas en el mes. El programa que se pide, deberá cargar una matriz **cant**, de orden **m*n**, en la que cada fila represente un vendedor, cada columna un artículo, y cada componente **cant[i][j]** almacene la cantidad del artículo j vendida por el vendedor i.*

Se pide emitir un listado con las cantidades totales realizadas por cada vendedor y las cantidades totales que se vendieron de cada artículo.

Discusión y solución: En el proyecto *F[16] Arreglos Bidimensionales* que acompaña a esta Ficha, se incluye el modelo *test02.py* en el que se resuelve el problema propuesto.

En problema se pide efectuar una operación típica en el procesamiento de matrices: la *totalización por filas y/o columnas*. Usaremos una matriz *cant* en la que cada fila represente un vendedor, y cada columna un artículo. En cada casillero se guarda el total de unidades que cada vendedor vendió de cada artículo. Por lo tanto, si se quiere saber cuántos artículos en total vendió un vendedor, sólo se deben acumular los componentes de la fila de ese vendedor. Si esa operación se realiza para cada fila, se está haciendo una *totalización por filas*. En forma similar se procede para los artículos, haciendo una *totalización por columnas*. En el primer caso, se hace un proceso por *fila creciente* acumulando cada componente. Y en el segundo, es un proceso por *columna creciente*, también acumulando cada casillero.

La función *totales_por_vendedor()* realiza una *totalización por filas* de la matriz, acumulando los valores de cada fila y mostrándolos por consola de salida a medida que termina de acumular cada fila. Como siempre, en el proceso de *totalización por filas* que desarrolla la función, el ciclo que recorre las filas va por fuera, y el que recorre las columnas va por dentro:

```
def totales_por_vendedor(cant):
    # totalización por filas...
    m, n = len(cant), len(cant[0])
    print()
    print('Cantidades vendidas por cada vendedor')
    for f in range(m):
        ac = 0
        for c in range(n):
            ac += cant[f][c]
        print('Vendedor', f, '\t- Cantidad total vendida:', ac)
```

Del mismo modo, la función *totales_por_articulo()* realiza una *totalización por columnas*, acumulando los valores de cada columna y mostrando los resultados por pantalla directamente antes de cambiar de columna. Otra vez, notar que la *totalización por columnas* se logra ubicando por fuera el ciclo de las columnas, y por dentro el de las filas, pero el orden de los índices en los corchetes para acceder a la matriz es siempre el mismo: primero el índice de fila y segundo el índice de columna:

```
def totales_por_articulo(cant):
    # totalización por columnas...
    m, n = len(cant), len(cant[0])
    print()
    print('Cantidades totales vendidas de cada artículo')
    for c in range(n):
        ac = 0
        for f in range(m):
            ac += cant[f][c]
        print('Artículo', c, '\t- Cantidad total vendida:', ac)
```

El programa completo se muestra a continuación:

```
def validate(inf):
    t = inf
    while t <= inf:
        t = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
        if t <= inf:
            print('Error: se pidio mayor a', inf, '... cargue de nuevo...')
    return t


def read(m, n):
    # crear y cargar por teclado una matriz... filas en orden creciente...
    cant = [[0] * n for f in range(m)]
    for f in range(m):
        for c in range(n):
            cant[f][c] = int(input('Valor [' + str(f) + '][' + str(c) + ']: '))
    return cant


def totales_por_vendedor(cant):
    # totalización por filas...
    m, n = len(cant), len(cant[0])
    print()
    print('Cantidades vendidas por cada vendedor')
    for f in range(m):
        ac = 0
        for c in range(n):
            ac += cant[f][c]
        print('Vendedor', f, '\t- Cantidad total vendida:', ac)


def totales_por_articulo(cant):
    # totalización por columnas...
```

```
    m, n = len(cant), len(cant[0])
    print()
    print('Cantidades totales vendidas de cada artículo')
    for c in range(n):
        ac = 0
        for f in range(m):
            ac += cant[f][c]
        print('Artículo', c, '\t- Cantidad total vendida:', ac)


def test():
    print('Cantidad de vendedores...')
    m = validate(0)

    print('Cantidad de artículos...')
    n = validate(0)

    print('Cargue las cantidades de artículos por vendedor...')
    cant = read(m, n)

    totales_por_vendedor(cant)
    totales_por_articulo(cant)


if __name__ == '__main__':
    test()
```

4.] Matrices de conteo y/o acumulación.

Así como en muchas situaciones prácticas se puede usar un arreglo unidimensional de forma que cada casilla actúe como un contador o como un acumulador (usándolo entonces como un *vector de conteo* o un *vector de acumulación*), también es posible que en ciertas ocasiones se deba emplear una matriz de contadores o de acumuladores. En algunos problemas se necesita seleccionar un contador entre muchos posibles, pero esa selección debe hacerse en base a dos variables que actúan como índices. El siguiente problema es un ejemplo típico:

Problema 44.) *Se desea almacenar en dos arreglos paralelos la información de los **n** clientes de una compañía de viajes que adquirieron algún viaje con esa empresa. En el primer arreglo se almacena en cada casillero un número entre 0 y 4 que indica el destino del viaje, y en el segundo arreglo se almacena otro número pero ahora entre 0 y 2 que indica la forma de pago que usó ese cliente. Se desea saber cuántos clientes viajaron a cada destino posible usando cada forma de pago disponible (es decir: cuántos clientes que viajaron al destino 0 usaron la forma de pago 0; cuántos clientes que viajaron al destino 0 usaron la forma de pago 1, y así sucesivamente. En total, se necesitan entonces 5*3 = 15 contadores, pues los destinos posibles son 5, y las formas de pago posibles son 3).*

Discusión y solución: En el proyecto *F[16] Arreglos Bidimensionales* que acompaña a esta Ficha, el modelo *test03.py* resuelve el problema propuesto.

Para resolver el conteo pedido en este ejercicio, se usa el concepto de *matriz de conteos*. Una *matriz de conteos* es análoga a un *vector de conteos*, con la diferencia que en la matriz de conteos se requieren *dos índices* para seleccionar el casillero que hará las veces de contador [1].

En este ejercicio, la matriz de conteos es usada para contar cuántos clientes que viajan a cada destino posible, usaron cada forma de pago posible. Como las formas de pago

disponibles son tres, y los destinos son cinco, hay entonces un total de quince combinaciones, cada una de las cuales requiere un contador. Como cada uno de los quince contadores debe seleccionarse con dos códigos (uno para la forma de pago y otro para el destino), se evidencia la *naturaleza bidimensional* del proceso de conteo.

Simplemente, se define una matriz de 5*3 elementos, para que cada uno de ellos funcione como un contador. Cada fila representa un destino de viaje y cada columna representa una forma de pago. En este programa, la matriz *conteo* se crea en la función *count()*, y luego para cada cliente se toma su destino de viaje (del primer arreglo) y la forma de pago que usó (del segundo arreglo). Ambos valores se usan como índices para acceder en forma directa al casillero correspondiente en la matriz, y se procede a contar en ese casillero:

```
def count(destinos, formas):
    conteo = [[0] * 3 for f in range(5)]

    n = len(destinos)
    for i in range(n):
        f = destinos[i]
        c = formas[i]
        conteo[f][c] += 1

    return conteo
```

La función *display_count()* muestra los resultados obtenidos, considerando sólo aquellos casilleros que hayan quedado con valor diferente de cero:

```
def display_count(conteo):
    filas, columnas = len(conteo), len(conteo[0])
    print()
    print('Conteo de clientes por destino y forma de pago')
    for f in range(filas):
        for c in range(columnas):
            if conteo[f][c] != 0:
                print('Destino', f, '\tForma', c, '\tCantidad:', conteo[f][c])
```

El programa completo se muestra a continuación. Dejamos el análisis del resto de los detalles para el estudiante.

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


def validate_range(inf, sup):
    n = inf - 1
    while n < inf or n > sup:
        n = int(input('Valor (entre ' + str(inf) + ' y ' + str(sup)+ '): '))
        if n < inf or n > sup:
            print('Se pidió entre', inf, 'y', sup, '... cargue de nuevo...')
    return n


def read(destinos, formas):
    n = len(destinos)
    for i in range(n):
        print('Destino del viaje -', end=' ')
        destinos[i] = validate_range(0, 4)
        print('Forma de pago -', end=' ')
```

```
        formas[i] = validate_range(0, 2)
        print()


def display(destinos, formas):
    n = len(destinos)
    print('Datos de los clientes registrados:')
    for i in range(n):
        print('Destino[', i, ']: ', destinos[i], sep='', end=' - ')
        print('Forma de pago:', formas[i])


def count(destinos, formas):
    conteo = [[0] * 3 for f in range(5)]

    n = len(destinos)
    for i in range(n):
        f = destinos[i]
        c = formas[i]
        conteo[f][c] += 1

    return conteo


def display_count(conteo):
    filas, columnas = len(conteo), len(conteo[0])
    print()
    print('Conteo de clientes por destino y forma de pago')
    for f in range(filas):
        for c in range(columnas):
            if conteo[f][c] != 0:
                print('Destino', f, '\tForma', c, '\tCantidad:', conteo[f][c])


def test():
    # cargar cantidad de clientes...
    print('Cantidad de clientes -', end=' ')
    n = validate(0)
    print()

    # crear y cargar los arreglos paralelos...
    destinos = n * [0]
    formas = n * [0]
    read(destinos, formas)
    print()

    # mostrar todos los clientes...
    display(destinos, formas)
    print()

    # contar por destino y forma de pago...
    conteo = count(destinos, formas)

    # mostrar por pantalla el listado...
    display_count(conteo)


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

5.] Diagonalización de matrices cuadradas.

Un interesante ejercicio de aplicación sobre matrices cuadradas [1] puede usarse para cerrar esta ficha. Mostramos directamente el enunciado y la solución propuesta.

Problema 45.) *Cargar por teclado una matriz cuadrada de números enteros, de orden n*n, y desarrollar las siguientes operaciones en ella:*

- **a.)** *Acumular los elementos ubicados en el triángulo superior de la matriz (es decir, los elementos ubicados por encima de la diagonal principal).*
- **b.)** *Determinar cuántos elementos de la diagonal principal valen 0(cero).*
- **c.)** *Determinar cuántos elementos ubicados en el triángulo inferior de la matriz (o sea, los elementos ubicados debajo de la diagonal principal) son pares.*
- **d.)** *Mostrar la matriz.*

Discusión y solución: En una matriz cuadrada pueden definirse tres zonas a partir de la diagonal principal (que en el gráfico siguiente se marca resaltada en gris):

Figura 2: La diagonal principal y los triángulos superior e inferior en una matriz cuadrada.

d	**s**	**s**	**s**	**s**
i	**d**	**s**	**s**	**s**
i	**i**	**d**	**s**	**s**
i	**i**	**i**	**d**	**s**
i	**i**	**i**	**i**	**d**

- ✓ Los elementos que pertenecen a la *diagonal principal* se marcan aquí con una letra *d*.
- ✓ Los elementos ubicados en el *triángulo superior* se marcan aquí con una letra *s*.
- ✓ Los elementos ubicados en el *triángulo inferior* se marcan aquí con una letra *i*.

El acceso a los elementos de la *diagonal principal* es directo si se observa que todos esos elementos tienen el *índice de fila igual al índice de columna*. La función *diagonal()* tiene el objetivo de recorrer esa diagonal y contar cada uno de los casilleros que contenga un cero. Sólo es necesario ***un ciclo***, y *repetir el índice en los dos pares de corchetes* al acceder a la matriz:

```
def diagonal(mat):
    cc, n = 0, len(mat)
    for f in range(1, n):
        if mat[f][f] == 0:
            cc += 1
    return cc
```

El acceso a los elementos del *triángulo superior* es un poco más complicado, pero resulta sencillo si se analiza lo siguiente: hay elementos del triángulo superior en todas las filas, *salvo en la última* (ver gráfico). Es decir que el ciclo que recorra las filas de la matriz no tiene necesidad de llegar hasta la fila *n–1* (que es la última). El ciclo para recorrer las filas comenzará colocando la variable *f* en cero, y continuará mientras *f* se mantenga menor que *n-1*. Y por otra parte, notemos que en la fila 0 el primer elemento del triángulo superior está en la columna 1. En la fila 2 el primero está en la columna 3... Puede verse fácilmente que en cada fila *f* el primer elemento del triángulo superior se encuentra en la columna *f+1*. Por eso, el ciclo que recorra las columnas (con variable de control *c*) no debe comenzar desde 0, sino desde *f+1*, y llegar en todos los casos hasta la última columna. Si los ciclos se ajustan de esta forma, solo se recorrerán los elementos que pertenecen al *triángulo superior*, sin tocar ni perder tiempo en los elementos que están fuera de él. La función *upper_triangle()* acumula los elementos del *triángulo superior* aplicando estos principios de recorrido:

```
def upper_triangle(mat):
    ac, n = 0, len(mat)
    for f in range(n-1):
        for c in range(f+1, n):
            ac += mat[f][c]
    return ac
```

Un análisis parecido permite ajustar los ciclos de recorrido para atravesar el *triángulo inferior*: todas las filas tienen elementos del *triángulo inferior*, salvo la *primera*. El ciclo para barrido de filas debe comenzar entonces con la variable de control *f* valiendo 1. Por otra parte, en la fila 1 el último elemento del triángulo inferior está en la columna 0. En la fila 2 el último está en la columna 1, y así sucesivamente: el ciclo que recorra las columnas (con variable de control *c*), debe comenzar entonces en 1 y proseguir mientras *c* se mantenga menor a *f*. La función *lower_triangle()* aplica estos principios para recorrer el *triángulo inferior* y contar los valores que sean pares del triángulo inferior:

```
def lower_triangle(mat):
    cp, n = 0, len(mat)
    for f in range(1, n):
        for c in range(0, f):
            if mat[f][c] % 2 == 0:
                cp += 1
    return cp
```

La visualización por pantalla de la matriz completa, pero de forma que cada fila aparezca a renglón seguido de la fila anterior, es realizada por la función *write()*, cuya estructura es muy simple:

```
def write(mat):
    n = len(mat)
    for f in range(n):
        print(mat[f])
```

El ciclo *for* de la función recorre todas las filas de la matriz *mat*, y en cada giro de ese ciclo sólo es necesario mostrar con *print()* la fila *f* completa (*print(mat[f])*), sin tener que usar otro ciclo para recorrerla: la fila *mat[f]* es ella misma una variable de tipo *list*, y la función *print()* ya está diseñada para mostrar correctamente toda una lista. El programa completo se ve a continuación (modelo *test04.py* en el proyecto *[F16] Arreglos Bidimensionales* que acompaña a esta ficha):

```
def validate(inf):
    t = inf
    while t <= inf:
        t = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
        if t <= inf:
            print('Error: se pidio > a', inf, '... cargue de nuevo...')
    return t


def read(n):
    # crear y cargar por teclado una matriz cuadrada...
    mat = [[0] * n for f in range(n)]
    for f in range(n):
        for c in range(n):
            mat[f][c] = int(input('Valor [' + str(f) + '][' + str(c) + ']: '))
    return mat


def write(mat):
    n = len(mat)
```

```
    for f in range(n):
        print(mat[f])


def upper_triangle(mat):
    ac, n = 0, len(mat)
    for f in range(n-1):
        for c in range(f+1, n):
            ac += mat[f][c]
    return ac


def lower_triangle(mat):
    cp, n = 0, len(mat)
    for f in range(1, n):
        for c in range(0, f):
            if mat[f][c] % 2 == 0:
                cp += 1
    return cp


def diagonal(mat):
    cc, n = 0, len(mat)
    for f in range(1, n):
        if mat[f][f] == 0:
            cc += 1
    return cc


def test():
    print('Orden de la matriz cuadrada...')
    n = validate(0)
    print()
    print('Cargue la matriz...')
    mat = read(n)
    print()
    print('Contenido de la matriz:')
    write(mat)
    r1 = upper_triangle(mat)
    r2 = diagonal(mat)
    r3 = lower_triangle(mat)
    print('Acumulación del triángulo superior:', r1)
    print('Cantidad de ceros en la diagonal:', r2)
    print('Cantidad de pares en el triángulo inferior:', r3)


if __name__ == '__main__':
    test()
```

Bibliografía

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

[2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: https://docs.python.org/3/.

[3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.