Ficha 20

Arreglos de Objetos

1.] Arreglos de registros/objetos (o *vectores de registros/objetos*) en Python.

En muchas aplicaciones será más útil almacenar registros/objetos completos en cada casillero de un arreglo unidimensional, en lugar de usar arreglos paralelos como mostramos en una ficha anterior. Esto es perfectamente válido en Python: un arreglo en Python puede contener elementos de cualquier tipo incluyendo *elementos que sean referencias a registros*. Esta técnica es efectivamente más cómoda que la de los arreglos paralelos, ya que en lugar de definir varios arreglos separados para almacenar en cada uno una parte de los valores que se necesitan, se define un único arreglo que contiene a todos los datos y con eso no solo se simplifica el código fuente sino que se ahorra esfuerzo cuando se tiene que enviar ese conjunto de datos como parámetro a una función, ya que en lugar de varios arreglos, se pasa sólo uno [3]. Un arreglo unidimensional de registros/objetos también suele designarse como un *vector de registros/objetos* (y en adelante, nos referiremos a ellos simplemente como *vectores de objetos*).

Supongamos que se tiene definida una clase *Estudiante* y que se desea almacenar varios objetos de esa clase en un arreglo. Suponga que se cuenta con la clase *Estudiante* ya definida y con una función constructora *__init()__* para crear y asignar los atributos a cada objeto de ese tipo:

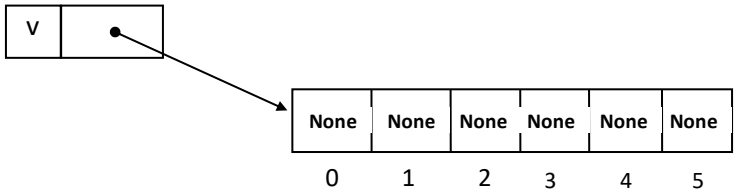
```
class Estudiante:
    def __init__(self, leg=0, nom='', pro=0):
        self.legajo = leg
        self.nombre = nom
        self.promedio = pro
```

Si se conoce de antemano la cantidad *n* de objetos a almacenar, una forma de crear el arreglo pedido consiste en declarar primero la referencia a ese arreglo, reservando *n* casilleros que pueden comenzar valiendo cualquier valor (por ejemplo, *None*): lo importante no es qué valor comience asignado en cada casillero (puesto que este valor puede cambiarse más tarde incluso por uno de otro tipo), *sino que se tengan efectivamente n casilleros* [1]:

```
n = 6
v = n * [None]
```

La instrucción anterior crea un arreglo de *n = 6* componentes con valor inicial *None*. La idea es que en cada uno se almacene luego una referencia a un objeto de tipo *Estudiante*, aunque todavía no se han creado esos objetos. El aspecto que tendría este arreglo en este momento en memoria sería como el que se muestra a continuación:

Figura 1: Un arreglo con *n = 6* casilleros con valor inicial *None*.



A diferencia de un arreglo que contenga valores de tipo simple, un arreglo de referencias todavía no contiene elementos listos para usar (a menos que el programador quiera efectivamente usar los valores *None*...): deben crearse los objetos que serán apuntados desde cada casilla del arreglo, y recién entonces comenzar a usar esos objetos. Por ejemplo, en este caso, podría hacerse algo como:

```
for i in range(n):
    v[i] = Estudiante()
```

lo cual haría que cada casilla del arreglo contenga ahora una referencia a un objeto de tipo *Estudiante*, pero con todos sus campos valiendo los *valores default que asigna la función constructora* (ya que al invocarla en el momento de crear cada objeto, no le hemos enviando ningún valor como parámetro formal). Los valores *None* ya no están en cada casillero, y fueron reemplazados por las direcciones de cada uno de los objetos. Una mejor forma de proceder podría ser que no solo se creen los objetos vacíos, sino que también se proceda a cargar por teclado los datos a almacenar en cada uno, y *luego se creen los objetos con sus atributos inicializados de forma definitiva*. La secuencia que sigue muestra la forma de hacerlo:

```
for i in range(n):
    leg = int(input('Legajo[' + str(i) + ']: '))
    nom = input('Nombre: ')
    pro = float(input('Promedio: '))
    print()
    v[i] = Estudiante(leg, nom, pro)
```

Está claro que puede lograrse el mismo resultado dejando los valores default de la función constructora *__init()__*, y luego *creando y asignando directamente los campos en cada casillero*. Lo anterior es equivalente a:

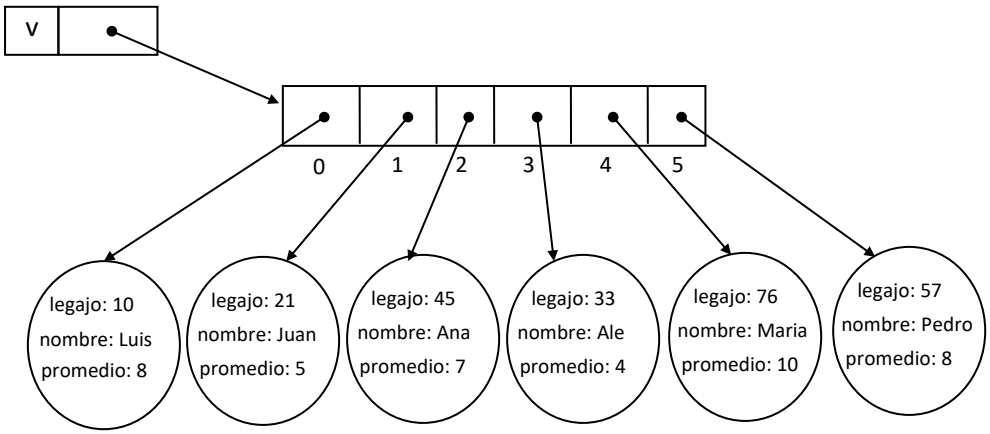
```
for i in range(n):
    v[i] = Estudiante()
    v[i].legajo = int(input('Legajo[' + str(i) + ']: '))
    v[i].nombre = input('Nombre: ')
    v[i].promedio = float(input('Promedio: '))
    print()
```

Como cada casillero ***v[i]*** del arreglo es un objeto de tipo *Estudiante*, entonces el acceso a cada campo de ***v[i]*** se hace con el operador punto combinado con el propio identificador ***v[i]***: *v[i].legajo* permite acceder al campo *legajo* del registro ubicado en *v[i]*, y en forma similar ocurre para el resto de los atributos.

Las dos secuencias anteriores crean diferentes objetos y los asignan en distintas casillas del arreglo, que podría verse ahora como se ve en la figura de la página siguiente (suponga que los datos cargados por teclado fueron efectivamente los que se ven allí).

Solo cuando cada objeto haya sido creado se puede recorrer el arreglo y aplicar a cada uno alguna tarea o proceso. Recuerde que *v[i]* es la referencia que apunta al objeto en la casilla *i*, y por lo tanto algo como *v[i].legajo* accede al número de legajo almacenado en el objeto *v[i]*. Pero esto solo es válido si *v[i]* es efectivamente un objeto de tipo *Estudiante*: si se intenta acceder a un campo desde una referencia que no apunta a un objeto de ese tipo, el programa se interrumpirá lanzando un mensaje de error por campo inexistente.

Figura 2: Un arreglo con *n = 6* referencias a registros ya creados de tipo *Estudiante*.



Finamente, el arreglo se usa como se usaría a cualquier arreglo: si se sabe qué clase de objetos se almacenaron dentro de él, se podrá cargarlos por teclado, visualizarlos, ordenarlos, efectuar búsquedas, etc. Analice el siguiente ejercicio que completa el ejemplo que hemos venido utilizando:

Problema 47.) *Se desea almacenar en un arreglo la información de los n estudiantes que se registraron para participar de un curso de programación. Por cada estudiante se tiene su número de legajo, su nombre y su promedio en la carrera que cursa. Participarán del curso los estudiantes cuyo promedio sea mayor o igual a x, siendo x un valor cargado por teclado. Muestre los datos de los estudiantes que participarán del curso, pero ordenados de menor a mayor por número de legajo.*

Discusión y solución: El programa que resuelve este problema es el modelo *test01.py* (incluido en el proyecto *[F20] Arreglos de Objetos* que viene con esta Ficha), y es el siguiente:

```
class Estudiante:
   def __init__(self, leg=0, nom='', pro=0):
      self.legajo = leg
      self.nombre = nom
      self.promedio = pro

   def __str__(self):
      return 'Legajo: ' + str(self.legajo) + ' - Nombre: ' + self.nombre + ' - Promedio: ' + str(self.promedio)


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


def read(estudiantes):
   n = len(estudiantes)
   for i in range(n):
      leg = int(input('Legajo[' + str(i) + ']: '))
      nom = input('Nombre: ')
      pro = float(input('Promedio: '))
```

```
        print()
        estudiantes[i] = Estudiante(leg, nom, pro)


def sort(estudiantes):
    n = len(estudiantes)
    for i in range(n-1):
        for j in range(i+1, n):
            if estudiantes[i].legajo > estudiantes[j].legajo:
                estudiantes[i], estudiantes[j] = estudiantes[j], estudiantes[i]


def display(estudiantes, x):
    n = len(estudiantes)
    print('Estudiantes que harán el curso (tienen promedio >=', x, '):')
    for i in range(n):
        if estudiantes[i].promedio >= x:
            print(estudiantes[i])


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

    # crear un arreglo con n casilleros de valor indefinido...
    # ... se usará para almacenar luego las referencias a los Estudiantes...
    estudiantes = n * [None]

    # cargar el arreglo por teclado...
    print('\nCargue los datos de los estudiantes:')
    read(estudiantes)
    print()

    x = float(input('Promedio mínimo para poder hacer el curso: '))
    print()

    # ordenar de menor a mayor el arreglo, por legajo...
    sort(estudiantes)

    # mostrar por pantalla el listado...
    display(estudiantes, x)


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

La función *test()* define un arreglo referenciado por la variable *estudiantes*, que contendrá referencias a objetos de tipo *Estudiante* (clase que fue definida al inicio del módulo). Este arreglo se usará para almacenar los datos de todos los estudiantes que se hayan inscripto para hacer el curso. La misma función *test()* carga por teclado la cantidad total de alumnos *n*, y en base a ese valor crea el arreglo con *n* casilleros valiendo *None*. Recuerde que no se debería comenzar a usar el arreglo hasta crear o asignar objetos de tipo *Estudiante* en cada componente (cosa que hace en este caso la función *read()*).

La función *sort()* ordena el arreglo de acuerdo a los legajos de los estudiantes, de menor a mayor. Lo más relevante de esta función *es la condición para determinar si debe hacerse un intercambio o no*:

```
if estudiantes[i].legajo > estudiantes[j].legajo:
                estudiantes[i], estudiantes[j] = estudiantes[j], estudiantes[i]
```

Puede verse que en la condición se accede al campo o atributo *legajo* de cada objeto, y se comparan sus valores: ahora cada casillero contiene un objeto y ***no*** un valor *int*, por lo que comparar directamente *estudiantes[i]* con *estudiantes[j]* carece de sentido (¡y provocaría un error!). Además, si la condición es verdadera puede verse que se procede a intercambiar directamente las referencias a los objetos, *sin tener que hacer ese intercambio campo por campo*.

Finalmente, la función *display()* recorre el contenido del arreglo mostrando por consola los datos de los alumnos cuyo promedio sea mayor o igual al valor *x* que se tomó como parámetro. Como el arreglo está ordenado por legajo, los datos de los estudiantes que se muestren saldrán a su vez ordenados por legajo.

Analicemos ahora otro problema en el que se requiere gestionar un vector de objetos, El enunciado es el siguiente:

Problema 48.) *Una empresa dedicada a la producción de piezas de repuestos automotrices está desarrollando un programa para gestionar un arreglo con datos de los **n** insumos que componen a una pieza en particular (cargar n por teclado). Todo el vector se usa para representar una misma y única pieza, y todos los insumos registrados en el vector son parte de esa única pieza.*

Por cada insumo se tiene como dato su código (es un valor numérico que puede tomar valores de 1 a 20 ambos incluidos), nombre (una cadena de caracteres), valor (es el precio del insumo) y cantidad (es un número que indica la cantidad utilizada del insumo para fabricar la pieza).

Desarrollar un programa en Python controlado por un menú de opciones. Ese menú debe permitir gestionar las siguientes tareas:

a) Cargar por teclado el vector pedido, validando que el código de insumo esté efectivamente entre 1 y 20.

b) Mostrar el valor total de la pieza representada por el vector. Para esto debe acumular (sumar) el valor obtenido de multiplicar el precio por la cantidad de cada insumo del arreglo.

c) Mostrar sólo los insumos cuya cantidad sea menor a un valor x cargado por teclado. Este listado debe mostrarse ordenado de menor a mayor de acuerdo al código de los insumos.

d) Cargar por teclado un valor c, y para todo insumo cuya cantidad sea igual a 0 cambiar esa cantidad por el valor c. Mostrar los datos de los insumos que hayan sido modificados.

e) Desarrollar una función que reciba como parámetro un código de insumo y muestre por pantalla todos los datos del mismo si se encuentra en el vector. Informe con un mensaje si ese insumo no existe.

Discusión y solución: El programa completo puede verse en el proyecto *[F20] Arreglos de Objetos* que acompaña a esta Ficha. Dentro de ese proyecto, el módulo *insumos.py* define la clase *Insumo* para manejar objetos que representen insumos. La clase incluye su función constructora *__init__()* con un parámetro por cada atributo previsto en el objeto, y el ya conocido método *__str__()* que retorna una cadena con el contenido del objeto *self*, de forma que esa cadena ya tiene el formato adecuado para ser visualizada en pantalla si fuese el caso:

```
class Insumo:
    def __init__(self, cod=1, nom='', pre=0.0, cant=0):
        self.codigo = cod
        self.nombre = nom
        self.valor = pre
        self.cantidad = cant

    def __str__(self):
        r = ''
        r += '{:<15}'.format('Codigo: ' + str(self.codigo))
        r += '{:<30}'.format('Nombre: ' + self.nombre)
        r += '{:<18}'.format('Precio: ' + str(self.valor))
        r += '{:<15}'.format('Cantidad: ' + str(self.cantidad))
        return r
```

El método *__str__()* usa en forma directa los atributos que sean de tipo cadena y también convierte cada atributo numérico del objeto a una cadena de caracteres con la función predefinida *str()* que Python ya provee. Cada cadena así obtenida se une a otra cadena descriptiva con el operador + en forma similar a lo que se muestra a continuación:

```
'Codigo: ' + str(self.codigo)
```

En principio, una cadena así formada ya estaría en condiciones de concatenarse a las de los otros campos para finalmente retornar la cadena completa. Sin embargo, para lograr un ajuste más fino en cuanto a la justificación hacia la izquierda y el espaciado entre valores dentro de la cadena final, se está usando aquí el método *format()* incluido dentro de la clase *str* que representa al conjunto de cadenas de caracteres en Python [1].

El método *format()* es invocado por una cadena de caracteres, y retorna esa misma cadena pero ajustada al formato indicado por los parámetros enviados a *format()* y por algunos elementos contenidos en la propia cadena que se designan como *campos de reemplazo*. Los campos de remplazo se componen de algún tipo de indicador de formato *encerrado entre dos llaves*. Veamos un ejemplo sencillo:

```
a, b =  2, 4,
cad = 'La suma de {0} + {1} es {2}'.format(a, b, a+b)
print(cad)
```

En el script anterior la cadena '*La suma de {0} + {1} es {2}*' es la cadena mediante la cual se invoca al método *format()*, y es por lo tanto la cadena cuyo formato se quiere ajustar. Dentro de esa cadena se pueden ver tres campos de reemplazo: {0}, {1} y {2}. A su vez, al método *format()* se le están pasando tres parámetros: a, b y a+b. Cuando el método es invocado, el campo de reemplazo {0} en la cadena es reemplazado por el valor (convertido a cadena) del *primer parámetro* que haya recibido (en este caso, el valor de la variable a). El segundo campo de reemplazo (o sea, {1}) se reemplaza con el valor del segundo parámetro enviado a *format()* (en este caso, el valor de b), y así en forma sucesiva hasta agotar todos

los campos de reemplazo. De esta forma el *print()* del final del script mostrará en pantalla la siguiente salida:

```
La suma de 2 + 4 es 6
```

Ahora bien: la cadena con la que se invoca al método *format()* puede estar completamente formada por un campo de reemplazo, que puede contener no solo números que referencien a un parámetro, sino otros símbolos (o *comodines*) que tienen significados diferentes según como se los agrupe. Veamos el siguiente ejemplo:

```
nom = 'Juan Perez'
edad = 21
cad1 = '{:<30}'.format('Nombre: ' + nom)
cad1 += '{:<10}'.format('Edad: ' + str(edad))
print(cad1)
```

Aquí, la cadena '*{:<30}*' con la que se invoca por primera vez al método está compuesta solamente por un campo de reemplazo, que contiene a su vez la secuencia '*:<30*'. Esta secuencia está indicando que la cadena que vaya a reemplazar a ese campo de reemplazo, debe ser alineada o *justificada* a la izquierda (lo que se indica con los caracteres '*:<*') y en total, la cadena completa debe ajustarse hasta ocupar 30 caracteres de largo (eso es lo que significa el 30 ubicado al final de la secuencia). Si la cadena original no llegase a completar 30 caracteres, los que falten hasta llegar a 30 hacia la derecha serán llenados con espacios en blanco. Hasta aquí la cadena *cad1* contiene una cadena como 'Nombre: Juan Perez ' con 30 caracteres de largo (el resaltado del fondo en color celeste es sólo un recurso de texto para reforzar la idea: obviamente, no será mostrado con ese color en la consola de salida) (y note que los últimos 12 caracteres de la derecha se completan con blancos).

Pero en el mismo script se está usando el operador += para concatenar (agregar al final) una segunda cadena a la que ya contenía la variable *cad1*. La cadena que será agregada es de la forma *'{:<10}'.format('Edad: ' + str(edad))* y esto indica que la nueva cadena debe también ajustarse a la izquierda, pero completarse hasta llegar a 10 caracteres. Esta segunda cadena contendrá entonces la secuencia 'Edad: 21 ' (con dos blancos al final). Pero como esta segunda cadena será agregada al final de la primera, y la primera ya estaba ajustada a la izquierda con hasta 30 caracteres, el resultado mostrado por el *print()* final será algo como:

```
'Nombre: Juan Perez                Edad: 21   '
```

Y puede notarse entonces que en total, la cadena completa que se formó tendrá entonces 30 + 10 = 40 caracteres de largo y espacios en blanco correctamente ubicados para facilitar el justificado hacia la izquierda (y por lo tanto el encolumnado) de todos los valores.

Si en lugar de '*{:<30}*' se escribiese '*{:>30}*' (con el signo > en lugar de <) el efecto sería el mismo, pero con la cadena *ajustada hacia la derecha* y rellenando con blancos a la izquierda. Y si se escribiese '*{:^30}*' (con ^ en lugar de < o >) el efecto sería una *cadena centrada* (con blancos de relleno en ambos márgenes). El siguiente script simple muestra un ejemplo de cada caso:

```
cad = '{:<30}'.format('Hola mundo')
print(cad)
# muestra: 'Hola mundo                    '

cad = '{:>30}'.format('Hola mundo')
print(cad)
# muestra: '                    Hola mundo'
```

```
cad = '{:^30}'.format('Hola mundo')
print(cad)
# muestra: '           Hola mundo           '
```

Con todo esto aclarado, es simple ver lo que hace nuestro método *__str__()* original:

```
def __str__(self):
    r = ''
    r += '{:<15}'.format('Codigo: ' + str(self.codigo))
    r += '{:<30}'.format('Nombre: ' + self.nombre)
    r += '{:<18}'.format('Precio: ' + str(self.valor))
    r += '{:<15}'.format('Cantidad: ' + str(self.cantidad))
    return r
```

Cada campo del objeto *self* con el que se invocó al método se convierte a cadena de caracteres (de ser necesario) y se forma por concatenación una sola gran cadena de un total de 15 + 30 + 18 + 15 = 78 caracteres de largo, con cada subcadena justificada a la izquierda y rellenada con blancos a la derecha (si fuese requerido). La cadena así formada es retornada por la función. Si los datos contenidos en un objeto *x* cualquiera de la clase *Insumo* fuesen:

```
x.nombre = 'Puerta derecha'
x.codigo = 10
x.precio = 1000
x.cantidad = 100
```

entonces el siguiente script:

```
print(x)
```

produciría esta salida en la consola:

```
'Codigo: 10     Nombre: Puerta derecha        Precio: 1000      Cantidad: 200   '
```

El módulo *insumos.py* no contiene ya otros elementos. El programa completo para resolver el problema está incluido en el archivo *test02.py* del mismo proyecto, y lo mostramos a continuación:

```
import insumos


def validate(inf):
    n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
    while n <= inf:
        n = int(input('Error... Se pidio > ' + str(inf) + '... Cargue de nuevo: '))
    return n


def validate_code(mn=1, mx=20):
    cod = int(input('Ingrese codigo (>= ' + str(mn) + ' y <= ' + str(mx) + '): '))
    while cod < mn or cod > mx:
        cod = int(input('Error... era >='+str(mn)+' y <='+str(mx)+'). De nuevo: '))
    return cod
```

```
def read(pieza):
    n = len(pieza)
    for i in range(n):
        nom = input('Nombre[' + str(i) + ']: ')
        cod = validate_code(1, 20)
        val = int(input('Precio: '))
        can = int(input('Cantidad: '))
        pieza[i] = insumos.Insumo(cod, nom, val, can)
        print()


def opcion1(pieza):
    print('Cargue los datos de los insumos de la pieza:')
    read(pieza)


def display_all(pieza):
    for insumo in pieza:
        print(insumo)


def opcion2(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Listado completo de insumos para la pieza')
    display_all(pieza)


def total_value(pieza):
    tv = 0
    for insumo in pieza:
        monto = insumo.valor * insumo.cantidad
        tv += monto
    return tv


def opcion3(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return
    print('Monto total en insumos para la pieza:', total_value(pieza))


def sort(pieza):
    n = len(pieza)
    for i in range(n-1):
        for j in range(i+1, n):
            if pieza[i].codigo > pieza[j].codigo:
                pieza[i], pieza[j] = pieza[j], pieza[i]
```

```
def display(pieza, x):
    print('Insumos con menos de', x, 'unidades en esta pieza: ')
    for insumo in pieza:
        if insumo.cantidad < x:
            print(insumo)


def opcion4(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    x = int(input('Cantidad x de unidades a controlar: '))
    sort(pieza)
    display(pieza, x)


def change_quantity(pieza, c):
    exists = False
    print('Insumos que pasaron de 0 unidades a', c, 'unidades en esta pieza: ')
    for insumo in pieza:
        if insumo.cantidad == 0:
            exists = True
            insumo.cantidad = c
            print(insumo)

    if not exists:
        print('No hay insumos con 0 unidades en esta pieza')


def opcion5(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese la nueva cantidad c para los insumos sin unidades...')
    c = validate(0)
    change_quantity(pieza, c)


def search(pieza, cod):
    for insumo in pieza:
        if insumo.codigo == cod:
            return insumo
    return None


def opcion6(pieza):
    if pieza[0] is None:
```

```
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese el codigo del insumo a buscar...')
    cod = validate_code(1, 20)
    ins = search(pieza, cod)
    if ins is not None:
        print(ins)
    else:
        print('No existe un insumo con ese codigo en la pieza')


def test():
    # cargar cantidad de insumos...
    print('Ingrese la cantidad de insumos en la pieza...')
    n = validate(0)

    # crear el arreglo (inicialmente vacio)...
    pieza = n * [None]

    opc = 0
    while opc != 7:
        print('\nMenu de opciones:')
        print('1. Cargar insumos de la pieza')
        print('2. Mostrar todos los insumos de la pieza')
        print('3. Mostrar el valor total de la pieza')
        print('4. Mostrar insumos con menos de x unidades')
        print('5. Cambiar cantidad en insumos con 0 unidades')
        print('6. Buscar un insumo por su codigo')
        print('7. Salir')

        opc = int(input('Ingrese su eleccion: '))

        if opc == 1:
            opcion1(pieza)

        elif opc == 2:
            opcion2(pieza)

        elif opc == 3:
            opcion3(pieza)

        elif opc == 4:
            opcion4(pieza)

        elif opc == 5:
            opcion5(pieza)

        elif opc == 6:
            opcion6(pieza)
```

```
    elif opc == 7:
        print("--- Programa finalizado ---")


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

El archivo en su primera línea contiene la inclusión del módulo *insumos.py* con la correspondiente instrucción *import*:

```
import insumos
```

La función *test()* del programa (ubicada casi al final del mismo) sirve como *punto de entrada*: es la función que se ejecuta desde el script principal, y contiene la creación inicial del arreglo con ***n*** componentes vacíos (o sea, *None*) y la gestión del menú principal:

```
def test():
    # cargar cantidad de insumos...
    print('Ingrese la cantidad de insumos en la pieza...')
    n = validate(0)

    # crear el arreglo (inicialmente vacio)...
    pieza = n * [None]

    opc = 0
    while opc != 7:
        print('\nMenu de opciones:')
        print('1. Cargar insumos de la pieza')
        print('2. Mostrar todos los insumos de la pieza')
        print('3. Mostrar el valor total de la pieza')
        print('4. Mostrar insumos con menos de x unidades')
        print('5. Cambiar cantidad en insumos con 0 unidades')
        print('6. Buscar un insumo por su codigo')
        print('7. Salir')

        opc = int(input('Ingrese su eleccion: '))

        if opc == 1:
            opcion1(pieza)

        elif opc == 2:
            opcion2(pieza)

        elif opc == 3:
            opcion3(pieza)

        elif opc == 4:
            opcion4(pieza)

        elif opc == 5:
            opcion5(pieza)

        elif opc == 6:
            opcion6(pieza)

        elif opc == 7:
            print("--- Programa finalizado ---")


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

La función *opción1()* activa la carga del arreglo desde el teclado, lo cual se hace con la función *read()*, que a su vez es auxiliada por las funciones *validate()* y *validate_code()* para validar la carga de los campos numéricos. No hay demasiado misterio en este grupo de funciones que replicamos aquí:

```
def validate(inf):
    n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
    while n <= inf:
        n = int(input('Error... Se pidio > ' + str(inf) + '... Cargue de nuevo: '))
    return n


def validate_code(mn=1, mx=20):
    cod = int(input('Ingrese codigo (>= ' + str(mn) + ' y <= ' + str(mx) + '): '))
    while cod < mn or cod > mx:
        cod = int(input('Error... era >='+str(mn)+' y <='+str(mx)+'). De nuevo: '))
    return cod


def read(pieza):
    n = len(pieza)
    for i in range(n):
        nom = input('Nombre[' + str(i) + ']: ')
        cod = validate_code(1, 20)
        val = int(input('Precio: '))
        can = int(input('Cantidad: '))
        pieza[i] = insumos.Insumo(cod, nom, val, can)
        print()


def opcion1(pieza):
    print('Cargue los datos de los insumos de la pieza:')
    read(pieza)
```

La función *opción2()* permite la visualización del contenido completo del arreglo en la pantalla. Lo primero que hace la función es comprobar si el arreglo efectivamente contiene algún dato o no, lo cual sale en forma directa comprobando el contenido del casillero 0: si el mismo es *None*, es porque el arreglo nunca fue cargado desde el teclado, y en ese caso la función termina avisando al usuario que el arreglo está vacío. Solo si el arreglo estuviese cargado, se invoca a la función *display_all()* y esta recorre y muestra el contenido del arreglo recurriendo en cada vuelta del ciclo iterador a un *print()* que muestra la conversión a cadena de cada objeto invocando automáticamente a nuestro ya analizado método *__str__()*:

```
def display_all(pieza):
    for insumo in pieza:
        print(insumo)


def opcion2(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Listado completo de insumos para la pieza')
    display_all(pieza)
```

La función *opción3()* activa el proceso de cálculo del monto total acumulado para la pieza completa representada por el arreglo. La función comienza también comprobando si el arreglo efectivamente contiene algún registro (cosa que de aquí en adelante harán las funciones que restan para cumplir con cada opción) y luego invoca a la función *total_value()*

para hacer la acumulación, que es muy simple [3]: se recorre el arreglo con un *for iterador*, y en cada vuelta de ese ciclo se calcula el monto de cada insumo multiplicando el precio (o valor) de ese insumo por la cantidad del mismo que haya usado, acumulando ese resultado. El valor final del acumulador es el monto total y se retorna al final de la función *total_value()*:

```
def total_value(pieza):
    tv = 0
    for insumo in pieza:
        monto = insumo.valor * insumo.cantidad
        tv += monto
    return tv


def opcion3(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Monto total en insumos para la pieza:', total_value(pieza))
```

La función *opción4()* se invoca para mostrar en pantalla un listado con los insumos cuya cantidad usada sea menor a un valor *x* que la propia función carga por teclado. Como el listado debe mostrarse ordenado de acuerdo al código de cada insumo, lo primero que se hace es entonces ordenar el arreglo completo mediante la función *sort()*, y luego se invoca a la función *display()* para mostrar sólo el subconjunto de registros que tengan un valor menor a *x* en el campo *cantidad* [3]:

```
def sort(pieza):
    n = len(pieza)
    for i in range(n-1):
        for j in range(i+1, n):
            if pieza[i].codigo > pieza[j].codigo:
                pieza[i], pieza[j] = pieza[j], pieza[i]


def display(pieza, x):
    print('Insumos con menos de', x, 'unidades en esta pieza: ')
    for insumo in pieza:
        if insumo.cantidad < x:
            print(insumo)


def opcion4(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    x = int(input('Cantidad x de unidades a controlar: '))
    sort(pieza)
    display(pieza, x)
```

La función *opción5()* tiene el objetivo de lanzar el proceso para cambiar el valor del campo *cantidad* en todos los insumos cuya cantidad original sea cero. El nuevo valor a asignar en ese campo se carga por teclado en la variable *c*, y luego se invoca a la función *change_quantity()* para recorrer el arreglo y producir los cambios. Esta última función recorre el arreglo con *for* iterador, y si el objeto analizado en la vuelta actual tiene su atributo *cantidad* valiendo cero, simplemente lo reemplaza por el valor *c*. Se usa una bandera llamada *exists* para saber (al final del ciclo) si efectivamente hubo objetos

modificados o no, y poder de esta forma, si fuese necesario, mostrar un mensaje avisando que no hubo cambios. Otra vez, para mostrar el listado de insumos se recurre automáticamente al método *__str__()*:

```
def change_quantity(pieza, c):
    exists = False
    print('Insumos que pasaron de 0 unidades a', c, 'unidades en esta pieza: ')
    for insumo in pieza:
        if insumo.cantidad == 0:
            exists = True
            insumo.cantidad = c
            print(insumo)

    if not exists:
        print('No hay insumos con 0 unidades en esta pieza')


def opcion5(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese la nueva cantidad c para los insumos sin unidades...')
    c = validate(0)
    change_quantity(pieza, c)
```

La función *opción6()* (la última) inicia el mecanismo de búsqueda de un insumo cuyo código coincida con el valor *cod* cargado por teclado. La búsqueda se hace mediante la función *search()*, que simplemente implementa un proceso de *búsqueda secuencial* [3]: si el registro buscado existe, se lo retorna completo. Y si no existe, la función *search()* retorna *None*. La función *opcion6()* chequea el valor retornado por *search()* y muestra el registro en caso de haberlo hallado, o un mensaje informando que ese insumo no existía (si ese fuese el caso). Para la visualización del registro, nuevamente se usa un print() combinado con el método *__str__()*:

```
def search(pieza, cod):
    for insumo in pieza:
        if insumo.codigo == cod:
            return insumo
    return None


def opcion6(pieza):
    if pieza[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese el codigo del insumo a buscar...')
    cod = validate_code(1, 20)
    ins = search(pieza, cod)
    if ins is not None:
        print(ins)
    else:
        print('No existe un insumo con ese codigo en la pieza')
```

Con esto concluye el análisis de la solución para este problema. Presentamos ahora otro problema similar, para desplegar y aplicar otras técnicas elementales de procesamiento de arreglos de registros:

Problema 49.) *Una asociación deportiva desea almacenar la información referida a sus n socios en un arreglo de registros (cargar n por teclado). Por cada socio, se pide guardar su*

número de identificación, su nombre, el arancel que paga cada mes y un código entre 0 y 9 para indicar el deporte que cada socio practica (suponiendo que por ejemplo, el 0 puede ser fútbol, el 1 básquet, y así hasta el código 9).

Se pide desarrollar un programa en Python controlado por un menú de opciones. Ese menú debe permitir gestionar las siguientes tareas a partir del arreglo pedido en el párrafo anterior:

1- *Cargar el arreglo pedido con los datos de los n socios. Sólo valide el código del deporte practicado, para asegurar que esté entre 0 y 9.*
2- *Mostrar los datos de todos los socios que paguen un arancel mayor a p, siendo p un valor que se carga por teclado.*
3- *Determinar y mostrar cuántos socios practican cada tipo de deporte posible (un contador para contar cuántos socios practican el deporte 0, otro para los que practican el deporte 1, etc.).*
4- *Mostrar todos los datos, ordenados de menor a mayor por número de identificación.*
5- *Determinar si existe algún socio cuyo nombre sea igual a x, siendo x una cadena que se carga por teclado. Si existe, cambiar el valor del campo arancel de forma de sumarle un valor fijo de 100 pesos, y mostrar todos los datos de ese socio por pantalla. Si no existe, informar con un mensaje.*

Discusión y solución: El programa que resuelve este problema es el modelo *test03.py* (incluido en el proyecto *[F20] Arreglos de Objetos* que viene con esta Ficha). Incluye el módulo *socios.py* con la definición de la clase *Socio*, su función constructora *__init()__* y método *__str__()*:

```
class Socio:
    def __init__(self, num, nom='', ara=0.0, cod=0):
        self.numero = num
        self.nombre = nom
        self.arancel = ara
        self.codigo = cod

    def __str__(self):
        r = ''
        r += '{:<15}'.format('Numero: ' + str(self.numero))
        r += '{:<30}'.format('Nombre: ' + self.nombre)
        r += '{:<18}'.format('Arancel: ' + str(self.arancel))
        r += '{:<15}'.format('Codigo: ' + str(self.codigo))
        return r
```

El programa completo para resolver el problema está incluido en el archivo *test03.py* del mismo proyecto, y se transcribe a continuación:

```
import socios

def validate(inf):
    n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
    while n <= inf:
        n = int(input('Se pidio mayor a ' + str(inf) + '... Cargue de nuevo: '))
    return n


def validate_code(mn=0, mx=9):
    cod = int(input('Ingrese codigo (>= ' + str(mn) + ' y <= ' + str(mx) + '): '))
    while cod < mn or cod > mx:
        cod = int(input('Se pidio >='+str(mn)+' y <='+str(mx)+'). De nuevo: '))
```

```
    return cod


def read(club):
    n = len(club)
    for i in range(n):
        nom = input('Nombre[' + str(i) + ']: ')

        print('Ingrese numero de socio...')
        num = validate(0)

        print('Ingrese arancel...')
        ara = validate(0)

        print('Ingrese código de deporte...')
        cod = validate_code(0, 9)

        club[i] = socios.Socio(num, nom, ara, cod)
        print()


def opcion1(club):
    print('Cargue los datos de los socios del club:')
    read(club)


def display(club, p):
    print('Listado de socios que pagan arancel mayor a', p, ':')
    for socio in club:
        if socio.arancel > p:
            print(socio)


def opcion2(club):
    if club[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese arancel para cotrolar...')
    p = validate(0)
    display(club, p)


def count(club):
    vc = 10 * [0]
    for socio in club:
        d = socio.codigo
        vc[d] += 1

    print('Cantidad de socios en cada deporte deporte disponible:')
    for i in range(10):
        if vc[i] != 0:
            print('Codigo de deporte:', i, 'Cantidad de socios:', vc[i])


def opcion3(club):
    if club[0] is None:
        print('No hay datos cargados en el arreglo...')
        return
    count(club)


def sort(club):
    n = len(club)
    for i in range(n-1):
        for j in range(i+1, n):
            if club[i].numero > club[j].numero:
                club[i], club[j] = club[j], club[i]
```

```
def display_all(club):
    print('Listado completo de socios del club:')
    for socio in club:
        print(socio)


def opcion4(club):
    if club[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    sort(club)
    display_all(club)


def search(club, nom):
    for socio in club:
        if socio.nombre == nom:
            return socio
    return None


def opcion5(club):
    if club[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    nom = input('Ingrese el nombre del socio a buscar: ')
    socio = search(club, nom)
    if socio is not None:
        socio.arancel += 100
        print('Socio encontrado... se incremento en $100 su arancel...')
        print('Datos modificados del socio:')
        print(socio)
    else:
        print('No existe un socio registrado con ese nombre')


def test():
    # cargar cantidad de socios...
    print('Ingrese la cantidad de socios del club...')
    n = validate(0)

    # crear el arreglo (inicialmente vacio)...
    club = n * [None]

    opc = 0
    while opc != 6:
        print('\nMenu de opciones:')
        print('1. Cargar socios')
        print('2. Mostrar socios que pagan arancel mayor a p')
        print('3. Conteo de socios por cada deporte')
        print('4. Listado ordenado de socios')
        print('5. Buscar socio y ajustar su arancel')
        print('6. Salir')

        opc = int(input('Ingrese su eleccion: '))

        if opc == 1:
            opcion1(club)

        elif opc == 2:
            opcion2(club)

        elif opc == 3:
            opcion3(club)
```

```
        elif opc == 4:
            opcion4(club)

        elif opc == 5:
            opcion5(club)

        elif opc == 6:
            print("--- Programa finalizado ---")


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

Las mayor parte de las tareas que deben llevarse a cabo para resolver este problema son similares a las que se expusieron para el problema anterior, por lo que confiamos en que el estudiante podrá realizar al análisis por su propia cuenta. Sin embargo, veremos con algún detalle el proceso lanzado por la función *opcion3()* para contar cuántos socios están registrados en cada uno de los 10 tipos de deportes que se ofrecen en el club[1].

Como esos deportes son 10 y cada uno está identificado con un número del 0 al 9, se puede aplicar la ya conocida técnica de *arreglo o vector de conteo* [3]. La función *count()* crea primero un arreglo *vc* de 10 elementos iniciados en cero, de forma que cada uno de esos elementos se use luego como un contador para cada deporte: la casilla *vc[0]* se usará para contar cuántos socios se registraron en el deporte 0, la casilla *vc[1]* se usará para contar los que se registraron en el deporte 1, y así con todas las demás. De esta forma, el conteo (que no es otra cosa que la determinación de la distribución de frecuencias de los socios por cada deporte), se realiza en forma directa: un ciclo iterador va tomando uno por uno los registros de los socios del arreglo, y se usa el valor del atributo *codigo* (que contiene el número que identifica al deporte elegido por cada socio) para acceder en forma directa al casillero del arreglo *vc* donde se debe contar. Así, si un socio tiene el atributo *codigo* con el valor 4, el siguiente segmento hace que se acceda a la casilla 4 del vector *vc* para incrementar en uno su valor:

```
d = socio.codigo
vc[d] += 1
```

Cuando todos los socios han sido contados de esta forma, se muestra convenientemente el vector *vc* en pantalla para producir el listado final. Para simplificar la salida, sólo se muestran los deportes que efectivamente hayan tenido al menos un socio registrado:

---

[1] A lo largo de la historia los deportes han fomentado la competencia entre clubes, organizaciones y naciones. Todo bien mientras la cosa quede el terreno de la civilización y la cultura, y mientras no se llegue a los extremos mostrados en la saga de películas *The Hunger Games* (*Los Juegos del Hambre*). En un futuro distópico, una sociedad devastada organiza cada año un terrible evento distractivo (y también disciplinario…) en el que los diferentes distritos están obligados a enviar dos jóvenes para que se enfrenten todos contra todos, y se maten unos a otros, hasta que solo uno sobreviva y se convierta en el admirado ganador. La saga está basada en la trilogía de novelas del mismo nombre de la autora *Suzanne Collins*, y en su versión cinematográfica se presentó en cuatro películas (todas protagonizadas por la inolvidable *Jennifer Lawrence*): *The Hunger Games* (*Los Juegos del Hambre*) de 2012, *Catching Fire* (*En Llamas*) de 2013, *Mockingjay – Part 1* (*Sinsajo – Parte 1*) de 2014 y *Mockingjay – Part 2* (*Sinsajo – Parte 2*) de 2015. Y está prevista una cercana precuela: *The Ballad of Songbirds and Snakes* (*La Balada de Pájaros Cantores y Serpientes*) para noviembre de 2023. Allí estaremos… ☺

```
def count(club):
    vc = 10 * [0]
    for socio in club:
        d = socio.codigo
        vc[d] += 1

    print('Cantidad de socios en cada deporte disponible:')
    for i in range(10):
        if vc[i] != 0:
            print('Codigo de deporte:', i, 'Cantidad de socios:', vc[i])


def opcion3(club):
    if club[0] is None:
        print('No hay datos cargados en el arreglo...')
        return
    count(club)
```

Se deja el resto de los procesos y funciones de este programa para ser analizados por el estudiante.

2.] Generación de un arreglo de objetos con contenido aleatorio ("generación automática").

En los dos problemas que hemos presentado en lo que va de esta Ficha, se ha usado un arreglo de objetos cuyo contenido inicial siempre era cargado desde el teclado. Pero en algunas ocasiones esa tarea es muy incómoda y engorrosa, ya que la carga de un conjunto de varios objetos con muchos campos lleva tiempo, y cuando se están haciendo pruebas sobre el funcionamiento del programa esa pérdida de tiempo es molesta.

Por ese motivo es interesante explorar alguna forma de generar el contenido de cada atributo de los objetos del arreglo en forma automática, recurriendo a procesos basados en generación de valores aleatorios (que ya hemos presentado en fichas anteriores). El siguiente problema sigue en la misma línea general que los ya mostrados en esta Ficha, pero se agrega una opción para permitir esa generación automática del contenido:

Problema 50.) *Una empresa concesionaria de peajes está desarrollando un programa que le permitirá registrar en un arreglo de registros los datos de los n vehículos que pasan por sus cabinas (cargar n por teclado). Por cada vehículo que pasa por una cabina se toman los siguientes datos: patente del vehículo (una cadena de caracteres), tipo de vehículo (un número entero), número de cabina (es un valor numérico entre 0 y 14 que indica por cual cabina ha pasado el vehículo) y el importe pagado.*

Desarrollar un programa en Python controlado por un menú de opciones. Ese menú debe permitir gestionar las siguientes tareas:

1- *Cargar por teclado el vector validando que el número de cabina sea efectivamente un número entre 0 y 14.*
2- *Generar en forma automática (con valores obtenidos en forma aleatoria) los contenidos de cada registro del arreglo.*
3- *Mostrar todos los datos ordenados por patente de menor a mayor.*
4- *Mostrar todos los datos de los vehículos que hayan pasado por la cabina x sin pagar peaje (es decir, los datos de los vehículos que pasaron por esa cabina y tienen registrado un importe o pago igual a 0). El valor x debe ser cargado por teclado.*

5- *Determine el importe acumulado que se registró por cada cabina (el importe acumulado por vehículos que pasaron por la cabina 0, lo mismo para la cabina 1, y así con las 15 cabinas). También determine la cantidad de vehículos que pasaron por cada cabina (un total de 15 contadores).*

6- *Cargue por teclado una patente p y determine cuántas veces la misma está registrada en el vector. Muestre todos sus datos cada vez que la encuentre, e indique al final cuántas veces aparece. Si esa patente no existe, informe con un mensaje.*

Discusión y solución: La solución está incluida en el mismo proyecto *[F20] Arreglos de Objetos* que acompaña a esta Ficha. Como de costumbre, la clase *Vehiculo*, su función *__init__()* y el método *__str__()* se incorporan en un módulo aparte, que se llama *vehiculos.py* en el mismo proyecto:

```
class Vehiculo:
    def __init__(self, pat, tip, cab, imp=0.0):
        self.patente = pat
        self.tipo = tip
        self.cabina = cab
        self.importe = imp

    def __str__(self):
        r = ''
        r += '{:<20}'.format('Patente: ' + self.patente)
        r += '{:<20}'.format('Tipo: ' + str(self.tipo))
        r += '{:<20}'.format('Cabina: ' + str(self.cabina))
        r += '{:<20}'.format('Importe: ' + str(self.importe))
        return r
```

El programa completo está desarrollado en el archivo *test04.py* del mismo proyecto, y lo mostramos a continuación:

```
import random
import vehiculos


def validate(inf):
    n = int(input('Valor (mayor a ' + str(inf) + ' por favor): '))
    while n <= inf:
        n = int(input('Error... Cargue de nuevo: '))
    return n


def validate_code(mn=0, mx=14):
    cod = int(input('Ingrese codigo (>= ' + str(mn) + ' y <= ' + str(mx) + '): '))
    while cod < mn or cod > mx:
        cod = int(input('Error... Cargue de nuevo: '))
    return cod


def read(peaje):
    n = len(peaje)
    for i in range(n):
        pat = input('Patente[' + str(i) + ']: ')

        print('Ingrese tipo de vehiculo...')
        tip = validate(0)

        print('Ingrese numero de cabina...')
        cab = validate_code(0, 14)

        print('Ingrese importe pagado...')
```

```
        imp = validate(-1)

        peaje[i] = vehiculos.Vehiculo(pat, tip, cab, imp)
        print()


def opcion1(peaje):
    print('Cargue los datos de los vehiculos:')
    read(peaje)


def generate(peaje):
    letras = ('ABC', 'BCD', 'CDE', 'EFG')

    n = len(peaje)
    for i in range(n):
        p1 = random.choice(letras)
        p2 = str(random.randint(100, 1000))
        pat = p1 + p2

        tip = random.randint(1, 20)
        cab = random.randint(0, 14)
        imp = random.randint(0, 50)

        peaje[i] = vehiculos.Vehiculo(pat, tip, cab, imp)

    print('Hecho... el arreglo ha sido generado...')


def opcion2(peaje):
    print('Se procede a la generacion automatica del arreglo... pulse <Enter>...')
    input()
    generate(peaje)


def sort(peaje):
    n = len(peaje)
    for i in range(n-1):
        for j in range(i+1, n):
            if peaje[i].patente > peaje[j].patente:
                peaje[i], peaje[j] = peaje[j], peaje[i]


def display_all(peaje):
    print('Listado completo de socios del peaje:')
    for v in peaje:
        print(v)


def opcion3(peaje):
    if peaje[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    sort(peaje)
    display_all(peaje)


def display(peaje, x):
    exists = False
    print('Listado de vehiculos que pasaron por la cabina', x, 'sin pagar peaje:')
    for v in peaje:
        if v.importe == 0 and v.cabina == x:
            exists = True
            print(v)

    if not exists:
        print('No hubo vehiculos que hayan pasado sin pagar por esa cabina')
```

```
def opcion4(peaje):
    if peaje[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    print('Ingrese numero de cabina para cotrolar...')
    x = validate_code(0, 14)
    display(peaje, x)


def count(peaje):
    vc = 15 * [0]
    va = 15 * [0]
    for v in peaje:
        d = v.cabina
        vc[d] += 1
        va[d] += v.importe

    print('Cantidad de vehiculos e importe acumulado por cada cabina:')
    for i in range(15):
        if vc[i] != 0:
            print('Cabina:', '{:<4}'.format(str(i)), end='')
            print('Cantidad de vehiculos:', '{:<6}'.format(str(vc[i])), end='')
            print('Total recaudado:', '{:<10}'.format(str(va[i])))


def opcion5(peaje):
    if peaje[0] is None:
        print('No hay datos cargados en el arreglo...')
        return
    count(peaje)


def search(peaje, pat):
    c = 0
    for v in peaje:
        if v.patente == pat:
            c += 1
            print(v)

    if c != 0:
        print('Cantidad de pasos registrados para ese vehiculo:', c)
    else:
        print('No esta registrado ese vehiculo')


def opcion6(peaje):
    if peaje[0] is None:
        print('No hay datos cargados en el arreglo...')
        return

    pat = input('Ingrese la patente a buscar: ')
    search(peaje, pat)


def test():
    print('Ingrese la cantidad de vehiculos a cargar...')
    n = validate(0)

    peaje = n * [None]

    opc = 0
    while opc != 7:
        print('\nMenu de opciones:')
        print('1. Cargar vehiculos en forma manual')
        print('2. Cargar vehiculos en forma automática')
```

```
        print('3. Listado de vehiculos ordenado por patente')
        print('4. Vehiculos que pasaron por cabina x sin pagar peaje')
        print('5. Conteo de vehiculos e importe acumulado (por cabina)')
        print('6. Listado de todos los pasos de un vehiculo')
        print('7. Salir')

        opc = int(input('Ingrese su eleccion: '))

        if opc == 1:
            opcion1(peaje)

        elif opc == 2:
            opcion2(peaje)

        elif opc == 3:
            opcion3(peaje)

        elif opc == 4:
            opcion4(peaje)

        elif opc == 5:
            opcion5(peaje)

        elif opc == 6:
            opcion6(peaje)

        elif opc == 7:
            print("--- Programa finalizado ---")


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

El programa incluye dos instrucciones *import*: una para habilitar el acceso al módulo predefinido *random* que contiene las funciones para manejo de números aleatorios [1] [2], y el segundo para habilitar el modulo *vehiculos.py* con la definición de la clase pedida por el enunciado. En general, todo el trabajo que se implementa en cada función puede ser perfectamente estudiado y entendido por los estudiantes, por lo que solamente nos concentraremos en la opción 2 del menú: la generación automática (en base a valores aleatorios) del arreglo de objetos.

La función *opcion2()* es invocada desde el ciclo de control del menú principal en la función *test()*. La función simplemente pone un mensaje aclaratorio en pantalla e invoca a la función *generate(peaje)* para que sea esta la que finalmente haga el trabajo de generar el contenido del arreglo *peaje*.

```
def generate(peaje):
    letras = ('ABC', 'BCD', 'CDE', 'EFG')

    n = len(peaje)
    for i in range(n):
        p1 = random.choice(letras)
        p2 = str(random.randint(100, 999))
        pat = p1 + p2

        tip = random.randint(1, 20)
        cab = random.randint(0, 14)
        imp = random.randint(10, 50)

        peaje[i] = vehiculos.Vehiculo(pat, tip, cab, imp)
```

```
    print('Hecho... el arreglo ha sido generado...')


def opcion2(peaje):
    print('Generacion automatica del arreglo... pulse <Enter>...')
    input()
    generate(peaje)
```

La generación de valores aleatorios para los atributos de tipo numérico no es un problema: sabemos que la función predefinida *random.randint(a, b)* retorna un número entero aleatoriamente elegido en el intervalo [a, b], con lo que es suficiente con invocarla tres veces en nuestro caso (una vez por cada uno de los campos *tipo* (un entero mayor a cero, y que para simplificar suponemos no mayor a 20), *cabina* (un entero entre 0 y 14) e *importe* (también para simplificar, lo suponemos entero entre 10 y 50). La secuencia que sigue genera esos valores convenientemente, y los almacena en tres variables locales *tip*, *cab* e *imp*:

```
tip = random.randint(1, 20)
cab = random.randint(0, 14)
imp = random.randint(10, 50)
```

Cuando el atributo cuyo valor se quiere generar en forma aleatoria es una cadena de caracteres, se pueden pensar muchísimas técnicas ingeniosas para hacerlo, y la elección final dependerá de las restricciones del enunciado del problema, la forma esperada de las cadenas a crear, y el gusto y la necesidad del programador. En este modelo mostramos una forma simple, que esperamos pueda servir como disparador de creatividad para que los estudiantes piensen e implementen sus propias técnicas.

Una cadena que represente una patente argentina (según el estándar de 1994) consta de tres letras mayúsculas y luego tres dígitos. Una forma de sencilla de empezar, consiste en armar una *tupla* con algunas cadenas de tres letras que puedan ser usadas como base para generar la primera parte de la patente, y luego seleccionar alguna de esas cadenas con la función *random.choice()*:

```
letras = ('ABC', 'BCD', 'CDE', 'EFG')
p1 = random.choice(letras)
```

El pequeño script anterior selecciona en forma aleatoria una de las cuatro cadenas contenidas en la tupla *letras*, y almacena la cadena elegida en la variable *p1*. Está claro que de esta forma todas las patentes generadas comenzarán con alguna de estas cuatro cadenas, pero se puede ampliar el rango simplemente aumentando el número de cadenas base en la tupla, o crear una tupla solo con letras y luego seleccionar aleatoriamente tres de esas letras para unirlas concatenadas en un sola variable (dejamos estas ideas para ser exploradas por los estudiantes).

La generación de la parte numérica de la patente puede hacerse nuevamente con *random.randint()*, y para asegurarnos que siempre vuelva un número con exactamente tres dígitos, el intervalo dentro del cual se debe seleccionar podría ser [100, 999], convirtiendo luego a cadena de caracteres ese número obtenido:

```
p2 = str(random.randint(100, 999))
```

La instrucción anterior deja en la variable *p2* una cadena aleatoriamente creada, compuesta por exactamente 3 dígitos. Finalmente, si la variable *p1* contiene la cadena base de tres letras y *p2* contiene la cadena de 3 dígitos, una instrucción como la que sigue permite obtener la cadena completa representando a la patente esperada, dejándola asignada en la variable *pat*:

```
pat = p1 + p2
```

Por lo tanto, la función *generate(peaje)* efectivamente llena el arreglo *peaje* con valores aleatorios en rangos correctos, incluidas la patente:

```
def generate(peaje):
    letras = ('ABC', 'BCD', 'CDE', 'EFG')

    n = len(peaje)
    for i in range(n):
        p1 = random.choice(letras)
        p2 = str(random.randint(100, 999))
        pat = p1 + p2

        tip = random.randint(1, 20)
        cab = random.randint(0, 14)
        imp = random.randint(10, 50)

        peaje[i] = vehiculos.Vehiculo(pat, tip, cab, imp)

    print('Hecho... el arreglo ha sido generado...')
```

Se deja el resto del programa para ser analizado por los estudiantes.

Bibliografía

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

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

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