Ficha 18

Registros

1.] Introducción.

Hemos analizado en fichas anteriores el concepto de *estructura de datos* que hemos definido como una variable capaz de contener varios valores al mismo tiempo. Sabemos que Python provee diversas formas de manejo de estructuras de datos y hemos usado algunas como las tuplas, las cadenas de caracteres y los rangos. Por razones relacionadas con el contexto y el enunciado de los problemas que hasta aquí se propusieron, en esas estructuras de datos que usamos hemos almacenado siempre elementos del mismo tipo: o bien sólo números enteros, o bien sólo cadenas de caracteres, o bien elementos de cualquier otro tipo que el programador hubiese necesitado.

Pero el hecho es que en algún momento el programador necesitará almacenar valores de *tipos diferentes* en una misma estructura de datos, para describir algún elemento o entidad del dominio del problema. Este tipo de situaciones existen y son (como siempre...) muy comunes.

A modo de ejemplo, supongamos que se requiere representar y almacenar datos de las distintas asignaturas de un plan de estudios en un programa simple. Supongamos que por cada asignatura se dispone de un código numérico de identificación y de una cadena de caracteres para su nombre.

Una primera idea simple (aunque no sea la idea final a emplear) es recordar que en Python una *tupla* es una secuencia de datos que pueden ser de tipos diferentes, de forma que se puede acceder a cada uno de esos datos por *vía de un índice*. En ese sentido, queda claro que una *tupla* podría usarse en forma *muy elemental* para representar una *asignatura*, ya que se podría emplear una tupla con dos elementos (uno para el código y otro para el nombre) por cada asignatura a describir, como se ve en el ejemplo que sigue [1] [2]:

```
a1 = 1, 'AED'
a2 = 2, 'PPR'
a3 = 3, 'TSB'
```

Las tres variables *a1*, *a2* y *a3* son *tuplas* que contienen (cada una de ellas) los datos de una asignatura, y *cada uno de esos datos puede accederse mediante su índice*. Por ejemplo, el siguiente script muestra en consola los datos de las tres asignaturas:

```
print('Asignatura 1 - Código:', a1[0], 'Nombre:', a1[1])
print('Asignatura 2 - Código:', a2[0], 'Nombre:', a2[1])
print('Asignatura 3 - Código:', a3[0], 'Nombre:', a3[1])
```

Sin embargo, el enfoque anterior tiene dos problemas: el primero es que desde el punto de vista de la *claridad del código fuente* el programador debe recordar en qué posición de cada tupla está cada valor de la asignatura y acceder a él mediante un índice numérico, que no es

tan descriptivo ni simple de recordar. El programador debe saber que el código de la materia lleva el índice 0 dentro de la tupla, y el nombre de la materia lleva el índice 1. El código fuente debería complementarse con innumerables comentarios aclaratorios que finalmente significan más trabajo y más cuidado para no cometer un error.

Y en este caso, el segundo problema es bastante más grave: en Python, una *tupla* es una *secuencia inmutable*… lo cual quiere decir que *una vez que se creó la tupla no pueden modificarse los valores almacenados en ella* [1]. Esto llevaría a situaciones claramente inaceptables: si el programador quisiese, por ejemplo, cambiar el nombre de una de las asignaturas, se produciría un error de intérprete y el programa se interrumpiría:

```
a3 = 3, 'TSB'
print('Asignatura 3 - Código:', a3[0], 'Nombre:', a3[1])

a3[1] = 'DLC'   # Error aquí!!!
print('Asignatura 3 - Código:', a3[0], 'Nombre:', a3[1])
```

El script anterior lanzaría un error en la tercera línea, similar al que mostramos aquí:

```
Traceback (most recent call last):
  File "C:/Ejemplo/registros.py", line 7, in <module>
    a3[1] = 'DLC'
TypeError: 'tuple' object does not support item assignment
```

Por supuesto, Python provee otros tipos de secuencias y estructuras de datos *mutables* (como las *listas*) que podrían usarse en lugar de las *tuplas* para evitar el problema anterior, pero seguiríamos teniendo el primer inconveniente: el programador tendrá que recordar en que casillero exacto de cada estructura está almacenado cada valor para deducir el índice de acceso.

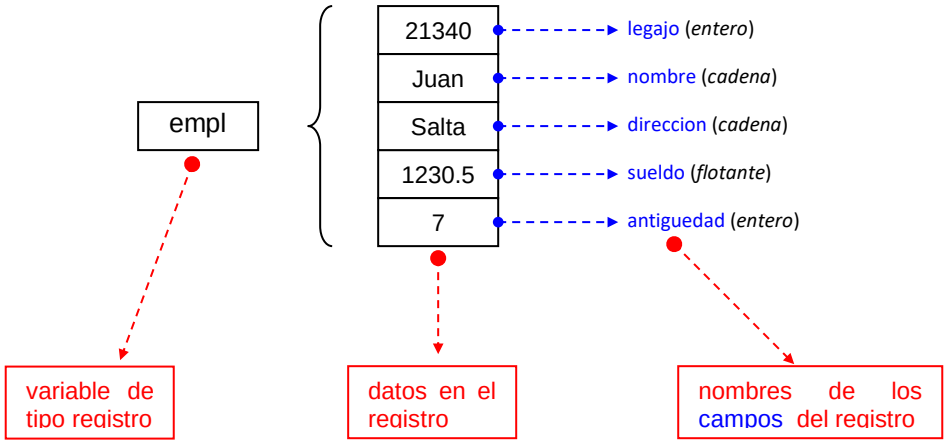
Claramente se requiere poder almacenar valores de tipos diferentes en una misma estructura de datos, pero de forma que la sintaxis y forma de gestión sea clara. Como vimos algunos de los enfoques que Python provee para lograr esto terminan siendo poco claros y por lo tanto, necesitamos una nueva estructura. Esa estructura es la que estudiaremos en esta Ficha y se designa con el nombre genérico de *registro*.

2.] Registros en Python.

Un *registro* es un conjunto *mutable* de valores que *pueden ser de distintos tipos*. Cada componente de un registro se denomina *campo* (o también *atributo*, dependiendo del contexto). Los *registros* son útiles para representar en forma clara a cualquier elemento (o *entidad* u *objeto*) del dominio o enunciado del problema que el programador necesite manejar y cuya descripción pudiera contener datos de tipos diferentes.

Por ejemplo, si se desea en un programa representar la información de un *empleado* de una empresa, puede usarse una variable *empl* de tipo *registro* con *campos* para el *legajo* (que sería un valor *de tipo entero*), el nombre (una *cadena de caracteres*), la dirección (otra *cadena de caracteres*), el *sueldo básico* (un valor de *coma flotante*) y la *antigüedad* de ese empleado en la empresa (que sería otro valor *entero*). El siguiente gráfico muestra una representación de la forma que podría tener tal variable [3]:

Figura 1: Esquema de una variable de tipo registro.



En el esquema anterior se supone el uso de una *variable registro* llamada *empl*, la cual consta de cinco *campos*, llamados *legajo*, *nombre*, *direccion*, *sueldo* y *antiguedad*. La importancia de los registros como estructuras de datos es evidente en el ejemplo anterior: sin registros en algunos lenguajes no sería posible almacenar en una misma variable todos los datos del empleado supuesto, ya que esos datos son todos de tipos diferentes, y en otros lenguajes se produciría mucha confusión en el acceso a cada campo, ya que el programador debería usar índices y saber de antemano en qué casilla quedó cada valor.

Note que no hay ningún problema si un programador quiere definir un registro que contenga *campos del mismo tipo*: un *registro* es una colección de variables llamadas *campos*, y esos campos *pueden* ser de tipos diferentes (pero ***no deben ser obligatoriamente*** de tipos distintos)

Para definir variables de tipo registro en un programa, lo común es proceder primero a declarar un nuevo nombre o identificador de *tipo de dato* para el registro. Con este nuevo tipo, se definen luego las variables. La declaración de un *tipo registro* se puede hacer en Python con la palabra reservada *class*, básicamente en la forma que se indica en el siguiente esquema (ver modelo *test01.py* - proyecto *[F18] Registros* que acompaña a esta ficha) [1]:

```
# declaración de un tipo Empleado como registro vacío
class Empleado:
    pass

# creación de variables de tipo Empleado
e1 = Empleado()
e1.legajo = 1
e1.nombre = 'Juan'
e1.direccion = 'Calle 1'
e1.sueldo = 10000
e1.antiguedad = 10

e2 = Empleado()
e2.legajo = 2
e2.nombre = 'Luis'
e2.direccion = 'Calle 2'
e2.sueldo = 20000
e2.antiguedad = 15

e3 = Empleado()
e3.legajo = 3
e3.nombre = 'Pedro'
```

```
e3.direccion = 'Calle 3'
e3.sueldo = 25000
e3.antiguedad = 20
```

En la forma que se trabajó en el esquema anterior, el identificador *Empleado* constituye un nuevo *tipo de dato* para el programa y en base a él se pueden definir variables que tendrán la estructura de campos o atributos que el programador necesite.

Las variables *e1*, *e2, y e3* del ejemplo anterior se crean como *registros* de tipo *Empleado* (también se dice que se crean como *instancias* de ese tipo) a través de la función constructora *Empleado()*, que está automáticamente disponible para tipos definidos mediante la palabra reservada *class*. Cada vez que se invoca a esa función, Python crea un registro vacío, sin ningún campo, y *retorna la dirección de memoria* donde ese registro vacío quedó alojado:

```
e1 = Empleado()
```

En el ejemplo anterior, la dirección retornada por la *función constructora* se asigna en la variable *e1*, que de allí en adelante se usa para acceder y gestionar el registro (se dice que *e1* está *apuntando al registro* o que está *referenciando al registro*).

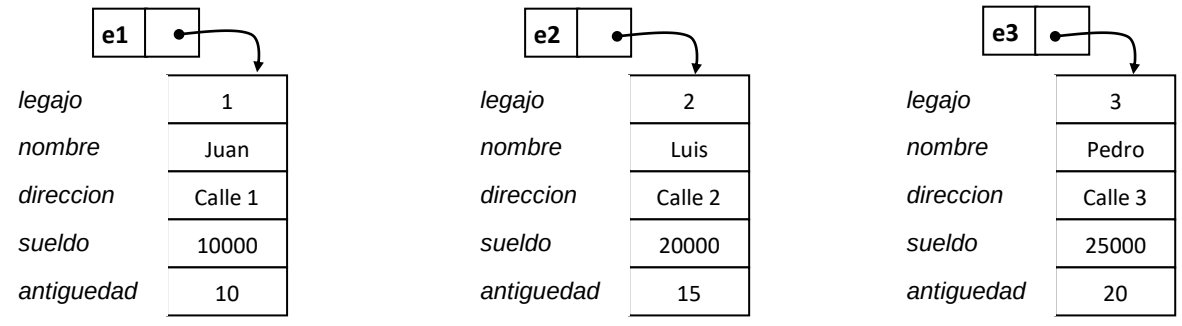
Como cada variable se crea inicialmente como un *registro vacío*, entonces luego el programador debe ir agregando campos (o atributos) a esas variables según lo necesite, *usando el operador punto (.)* para hacer el acceso:

```
e1 = Empleado()
e1.legajo = 1
e1.nombre = 'Juan'
e1.direccion = 'Calle 1'
e1.sueldo = 10000
e1.antiguedad = 10
```

Como se dijo, el acceso a los campos individuales de una variable *registro* se hace con el *operador punto (.)* en forma muy sencilla: se escribe primero el *nombre de la variable* registro (que contiene la dirección de memoria de ese registro), luego un *punto*, y finalmente el *nombre del campo* que se quiere agregar o acceder. Con esto se forma lo que se conoce como el *identificador del campo*, y a partir de aquí se opera con ese campo en forma normal, como se haría con cualquier variable común. En este caso, se están agregando cinco campos a la variable *e1*, llamados *legajo*, *nombre*, *direccion*, *sueldo* y *antiguedad*, cada uno asignado con el valor inicial que se requiera.

Luego de crearse las tres variables *e1*, *e2* y *e3* y luego de agregarse a cada una los campos citados, la memoria asignada a ellas podría verse como en el esquema de la *Figura 2*.

Figura 2: Esquema de memoria para tres registros de tipo *Empleado*.



Como se puede ver, cada variable apunta a un registro diferente de forma que cada uno contiene su propia copia de cada uno de los cinco campos, y cada campo en cada registro tiene su propio valor, que es independiente del valor que ese mismo campo tenga en los otros registros. Las siguientes instrucciones muestran en consola estándar el *legajo* y el *nombre* de cada uno de los tres empleados:

```
# mostrar legajo y nombre de cada empleado...
print('Empleado 1 - Legajo:', e1.legajo, '- Nombre:', e1.nombre)
print('Empleado 2 - Legajo:', e2.legajo, '- Nombre:', e2.nombre)
print('Empleado 3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre)
```

Además, como un registro es una estructura *mutable*, el valor de cualquier campo puede ser modificado cuando se necesite, recordando siempre que el acceso a un campo se realiza mediante el *operador punto*. Mostramos algunos ejemplos de operaciones válidas para los campos de los registros *e1*, *e2* y *e3* que ya hemos definido en los ejemplos anteriores [3]:

```
# algunas operaciones válidas...

# cambiar el legajo de e1...
e1.legajo = 4

# hacer que la antiguedad de e3 sea igual que la de e2...
e3.antiguedad = e2.antiguedad

# cargar por teclado el sueldo de e2...
e2.sueldo = float(input('Ingrese el nuevo sueldo del empleado 2: '))

# mostrar la direccion del empleado e1...
print('Direccion del empleado 1:', e1.direccion)

# sumar un año a la antiguedad de e3...
e3.antiguedad += 1
```

También recuerde que en Python las variables no quedan atadas a un tipo fijo y estático, y de hecho, nada impediría entonces que se cree una cuarta variable de tipo *Empleado* con una *estructura de campos diferente* a las tres que ya existen, como se ve en la variable *e4* que se muestra a continuación, en la cual *se ha agregado un campo extra para indicar el cargo del empleado* (ver modelo *test02.py* en el proyecto *[F18] Registros* que acompaña a esta Ficha) [1] [2]:

```
# creación de una variable Empleado con un campo adicional...
e4 = Empleado()
e4.legajo = 10
e4.nombre = 'Luis'
e4.direccion = 'Calle 4'
e4.sueldo = 50000
e4.antiguedad = 30
e4.cargo = 'Gerente'
```

En el ejemplo anterior, la variable *e4* se crea también como un registro de tipo *Empleado*, y se agregan en ella los mismos cinco campos que se habían incluido en las otras tres. Pero en la última instrucción se agrega para *e4* el campo *cargo*, que las otras tres no tenían: La variable *e4* (y *sólo la variable e4*) contendrá el campo *cargo* para asignar en él el nombre del puesto o cargo ocupado por el empleado. La siguiente secuencia de instrucciones muestra los legajos y los nombres de los cuatro empleados, más el cargo del cuarto:

```
# mostrar datos básicos...
print('E1 - Legajo:', e1.legajo, '- Nombre:', e1.nombre)
print('E2 - Legajo:', e2.legajo, '- Nombre:', e2.nombre)
print('E3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre)
print('E4 - Legajo:', e4.legajo, '- Nombre:', e4.nombre, '- Cargo:', e4.cargo)
```

Como las variables *e1*, *e2* y *e3* no contienen el campo *cargo*, se produciría un error de intérprete al intentar accederlo para consultarlo:

```
print('E3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre, '- Cargo:', e3.cargo)
```

La instrucción anterior produciría un error como el siguiente:

```
File "C:/Ejemplo/test02.py", line 42, in <module>
    print('E3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre, '- Cargo:', e3.cargo)
AttributeError: 'Empleado' object has no attribute 'cargo'
```

En general, la inicialización de un registro (así como cualquier otra operación) puede canalizarse por medio de funciones: no hay motivo para escribir largas secuencias de código fuente que hagan la misma tarea. En el siguiente modelo simple (*test03.py* – proyecto *[F18] Registros*) se crean los mismos tres registros de tipo *Empleado* que ya hemos visto, pero cada uno de ellos es inicializado mediante una función que hemos llamado *init()*, y luego cada uno de ellos es visualizado mediante otra función que hemos llamado *write()*:

```
__author__ = 'Cátedra de AED'


# declaración de un tipo registro vacío
class Empleado:
    pass


# una función para inicializar un registro de tipo Empleado
def init(empleado, leg, nom, direc, suel, ant):
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant


# una función para mostrar un registro de tipo Empleado
def write(empleado):
    print("\nLegajo:", empleado.legajo, end=' ')
    print("- Nombre:", empleado.nombre, end=' ')
    print("- Direccion:", empleado.direccion, end=' ')
    print("- Sueldo:", empleado.sueldo, end=' ')
    print("- Antiguedad:", empleado.antiguedad, end=' ')


# una funcion de prueba
def test():
    # creación de variables vacias de tipo Empleado...
    e1 = Empleado()
    e2 = Empleado()
    e3 = Empleado()

    # inicializacion de campos de las tres variables...
    init(e1, 1, 'Juan', 'Calle 1', 10000, 10)
```

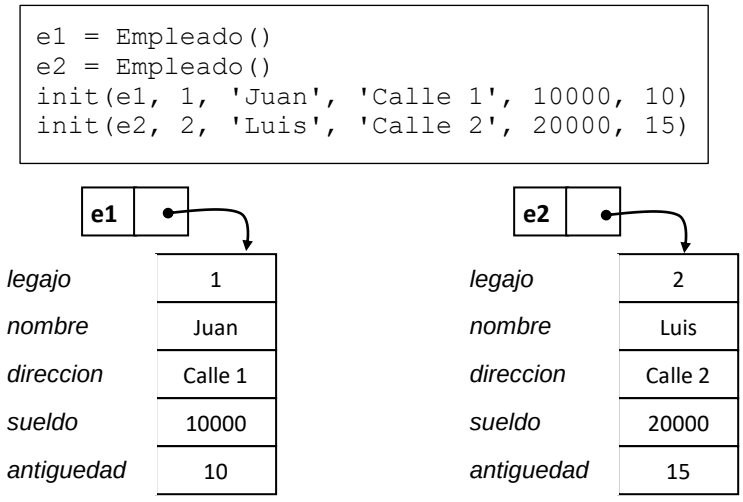
```
    init(e2, 2, 'Luis', 'Calle 2', 20000, 15)
    init(e3, 3, 'Pedro', 'Calle 3', 25000, 20)

    # visualizacion de los valores de los tres registros...
    write(e1)
    write(e2)
    write(e3)


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

Note que en este ejemplo simple, es la propia función ***init()*** la que finalmente "define" la estructura de campos del registro que entra como primer parámetro.

Un detalle que no debe pasarse por alto, es que como ya dijimos, en Python una variable de tipo *registro* contiene en realidad la *dirección de memoria* del registro asociado a ella (se dice la variable es *una referencia a ese registro*). Por lo tanto, si se asignan dos variables de tipo *registro* entre ellas (por ejemplo, dos registros de tipo *Empleado*) ***no se estará haciendo una copia*** del registro original, ***sino una copia de las direcciones de memoria***, haciendo que ambas variables queden apuntando *al mismo registro*.

A partir de aquí, entonces, el programador debe tener cuidado de entender bien lo que hace cuando opera con variables que son *referencias*: supongamos que las dos variables *e1* y *e2* apuntan a dos registros diferentes, ambos de tipo *Empleado*. El gráfico de memoria se vería como en la gráfica que sigue:

Figura 3: Dos referencias apuntando a registros diferentes.

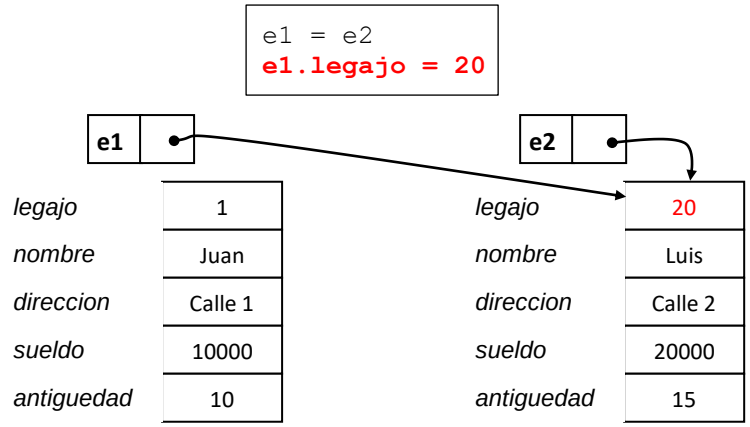


Si luego se hace una asignación como la siguiente:

```
e1 = e2
```

entonces *ambas* variables pasarán a apuntar o referenciar al *mismo* registro (el que originalmente era apuntado por *e2*). Por lo tanto, si se modifica el valor de algún campo para e1, se estará modificando *también* el registro referido por *e2*. La asignación *e1 = e2* ***no copia*** los contenidos del registro apuntado por *e2* hacia el registro apuntado por *e1*. Lo que copia es la *dirección contenida en e2* en la variable *e1*:

Figura 4: Dos referencias apuntando al mismo registro.



El registro apuntado originalmente por *e1* queda *des-referenciado*: esto significa que el programa ha perdido la dirección de memoria de ese registro y ya no hay forma de volver a utilizarlo. En algunos lenguajes (como C++), esta situación implica un error por parte del programador, ya que el registro des-referenciado sigue ocupando memoria aunque ya no pueda ser accedido.

Pero en Python (como en Java y otros lenguajes) eso no es necesariamente un error, pues en Python existe un sistema de *recolección de residuos* de memoria que se encarga de chequear la memoria en busca de variables "perdidas" en tiempo de ejecución. Cuando un programa se ejecuta, *en otro hilo de ejecución paralelo* corre también un proceso llamado *garbage collector*, que se encarga de los elementos des-referenciados, sin que el programador deba preocuparse por ellos.

El hecho ya citado de que una variable registro mantiene una referencia al registro, también hace que al enviar como parámetro una variable registro a una función, la misma pueda usarse para modificar el ***contenido*** del registro (aunque no puede cambiarse la dirección contenida en el parámetro). La función *init()* que ya hemos visto, hace justamente eso: toma como parámetro una variable *empleado* apuntando a un registro *ya creado*, y procede a modificar *su contenido* para inicializarlo:

```
def init(empleado, leg, nom, direc, suel, ant):
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant
```

Si la función *init()* intentase cambiar el valor de la variable *empleado* como se muestra en el modelo siguiente:

```
def init(empleado, leg, nom, direc, suel, ant):
    empleado = Empleado()
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant
```

entonces dentro de la función la variable *empleado* estaría apuntando a un registro que efectivamente sería inicializado con los cinco campos asignados. Pero al terminar de ejecutarse la función, la variable local *empleado* se destruye, y con eso el registro que se había creado queda des-referenciado. La variable registro que hubiese sido enviada como parámetro actual quedará sin cambios y seguirá apuntando al registro que apuntaba originalmente. Con estas consideraciones, el siguiente programa (modelo *test04.py* en el proyecto *F[18] Registros]*) produce un error al ejecutarse:

```
__author__ = 'Cátedra de AED'


# declaración de un tipo registro vacío
class Empleado:
    pass


# una función para inicializar un registro de tipo Empleado
def init(empleado, leg, nom, direc, suel, ant):

    # cuidado con esta línea...
    empleado = Empleado()

    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant


# una función para mostrar por consola un registro de tipo Empleado
def write(empleado):
    print("\nLegajo:", empleado.legajo, end=' ')
    print("- Nombre:", empleado.nombre, end=' ')
    print("- Direccion:", empleado.direccion, end=' ')
    print("- Sueldo:", empleado.sueldo, end=' ')
    print("- Antiguedad:", empleado.antiguedad, end=' ')


# una funcion de prueba
def test():
    e1 = Empleado()
    init(e1, 1, 'Juan', 'Calle 1', 10000, 10)
    write(e1)


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

El error se produce cuando al regresar de la función *init()* se invoca a la función *write()* para mostrar el registro. La función *write()* intentará mostrar los campos de la variable *e1*, pero debido al mecanismo antes descripto, *esa variable no tiene ningún campo*: fue creada en la función *test()* como registro vacío, luego se envió como parámetro a *init()*, la cual intentó cambiar la dirección que contenía por otra. Como eso no es posible debido al mecanismo de parametrización por valor, la variable *e1* volvió a *test()* exactamente con el mismo valor que tenía antes de invocar a *init()*: la dirección de un registro vacío. Y eso provoca el fallo en *write()*…

```
Traceback (most recent call last):
  File "C:/Ejemplo/test04.py", line 37, in <module>
    test()
  File "C:/Ejemplo/Fuentes/[F14] Registros/test04.py", line 32, in test
    write(e1)
  File "Ejemplo/test04.py", line 21, in write
    print("\nLegajo:", empleado.legajo, end=' ')
AttributeError: 'Empleado' object has no attribute 'legajo'
```

Obviamente, los problemas anteriores se resuelven de inmediato en forma simple, haciendo que nuestra la función *init() cree el registro con la función constructora, lo inicialice y retorne el registro* que acaba de crear e inicializar, en forma similar a lo siguiente:

```
class Empleado:
    pass

def init(leg, nom, direc, suel, ant):
    empleado = Empleado()
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant
    return empleado
```

Si la función *init()* está definida así, entonces la creación de un objeto puede hacerse de forma más simple y natural, como se ve en la función *test()* que sigue:

```
def test():
    e1 = init(1, 'Juan', 'Calle 1', 10000, 10)
    write(e1)
```

Note que esta nueva versión de nuestra función *init()* ahora no necesita tomar como parámetro el registro cuyos campos se desea crear, ya que la misma función crea ese registro y lo completa campo a campo, para finalmente retornarlo.

3.] Creación e inicialización de registros mediante constructores.

Como vimos, la palabra reservada *class* permite declarar un nuevo tipo de datos a modo de un conjunto capaz de contener una colección de variables (los *campos*). Pero la novedad es que el conjunto o tipo definido con *class* puede contener *funciones* además de *variables* o *campos*.

Antes de continuar conviene aclarar algunos elementos de terminología: el concepto de *registro* como agrupamiento de variables de tipos que pueden ser diferentes es propio del contexto de la *Programación Estructurada* (que abrevamos como *PE*), y muchos lenguajes que soportan ese paradigma disponen de palabras reservadas específicas para definir *registros* (tales como *record* en Pascal o *struct* en C). Y en el contexto de la *PE* un *registro* es una colección de variables que pueden ser de tipos diferentes, pero ***un registro no puede contener funciones***.

Sin embargo, la posterior llegada de la *Programación Orientada a Objetos* (abreviada como *POO*) aportó nuevos conceptos y herramientas, entre las que aparece nítidamente la noción

de *clase*. En la *POO*, una *clase* es un tipo de datos muy similar al *registro* de la *PE*, pero con una primera diferencia esencial: ***una clase puede contener funciones además de campos***.

Los elementos que en un *registro* se llaman *campos* (o sea, las *variables* definidas dentro del *registro*), reciben el nombre de *atributos* cuando se definen en una *clase*. Y si una *función* se define dentro de una *clase*, pasa a designarse como un *método* (en lugar de una *función*).

Finalmente (y como ya vimos), las variables que se definen en base a un tipo registro se llaman ellas mismas *registros* (o también *instancias*), pero las variables que se definen en base a una clase se suelen designar como *objetos* (aunque aquí también cabe la designación de *instancias*)

En Python, la palabra reservada *class* está claramente pensada para definir *clases* (y no *registros*). Obviamente, un programador que aún no tiene conocimientos de *POO* puede usar la palabra *class* para definir *registros* simplemente incluyendo sólo campos o atributos en la clase pero sin incluir funciones o métodos. Y ese ha sido el criterio que hemos seguido en a lo largo de esta Ficha: usar versiones simplificadas de clases (con atributos pero sin métodos) para emular el concepto de registro.

Por supuesto esta técnica para crear e inicializar un registro es válida, pero el hecho es que no hay motivo real para desaprovechar la potencia del uso clases y los de métodos en esas clases si el lenguaje utilizado provee el soporte. Y Python provee clases… no registros. Por lo tanto, veamos cuál es la forma de usar correctamente el recurso.

Recuerde: una clase puede pensarse como un registro que además de contener campos (o atributos), puede contener funciones (o métodos). Y uno de los métodos más importantes que una clase puede contener se designa como *método constructor* (o simplemente, un *constructor*). El objetivo de un constructor, es la creación y la inicialización de una instancia. En Python, un método constructor debe llamarse específicamente *__init__()* y como dijimos, debe definirse dentro del ámbito de la clase. El programa siguiente (*test05.py – Proyecto [F18] Registros*) es el mismo que ya vimos para crear instancias de tipo *Empleado*, pero ahora modificado para que la clase *Empleado* contenga el constructor *__init__()* ya citado:

```
__author__ = 'Cátedra de AED'


# declaración de una clase con un constructor...
class Empleado:
    # un constructor dentro de la clase Empleado...
    def __init__(self, leg, nom, direc, suel, ant):
        self.legajo = leg
        self.nombre = nom
        self.direccion = direc
        self.sueldo = suel
        self.antiguedad = ant


# una función para mostrar un objeto o instancia de la clase Empleado
def write(empleado):
    print("\nLegajo:", empleado.legajo, end=' ')
    print("- Nombre:", empleado.nombre, end=' ')
    print("- Direccion:", empleado.direccion, end=' ')
    print("- Sueldo:", empleado.sueldo, end=' ')
    print("- Antiguedad:", empleado.antiguedad, end=' ')
```

```
# una funcion de prueba
def test():
    # creación e inicialización de objetos de la clase Empleado...
    e1 = Empleado(1, 'Juan', 'Calle 1', 10000, 10)
    e2 = Empleado(2, 'Luis', 'Calle 2', 20000, 15)
    e3 = Empleado(3, 'Pedro', 'Calle 3', 25000, 20)

    # visualizacion de los valores de los tres objetos...
    write(e1)
    write(e2)
    write(e3)


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

Todo método de una clase debe definir un primer parámetro formal llamado *self* que represente al objeto o instancia (o registro) sobre el cual se aplica el método. Si ese método es el constructor (*__init__()*) entonces *self* es el objeto o registro que se está creando e inicializando.

Si la clase *Empleado* tiene definido el constructor que hemos mostrado, entonces la siguiente instrucción crea un objeto o registro *e1* de tipo *Empleado*, y lo inicializa con los valores enviados como parámetros actuales:

```
e1 = Empleado(1, 'Juan', 'Calle 1', 10000, 10)
```

La parte derecha de la asignación anterior es una invocación al constructor. Aun cuando el nombre *__init__()* no está presente en esa invocación, Python interpreta que si el nombre de una clase se usa a la derecha de una expresión de asignación y se le pasan parámetros, entonces se está invocando *implícitamente* al constructor de esa clase. Note además que no es necesario enviar como parámetro al propio objeto *e1* que se quiere crear: Python también interpreta que si se invoca a un constructor, el objeto que se construye es enviado automáticamente como parámetro a ese constructor, y por si fuese poco, el constructor automáticamente retorna la dirección del objeto creado (sin necesidad de incluir un *return*) razón por la cual el retorno del constructor es asignado en la variable *e1*.

En otras palabras: la expresión anterior invoca al constructor *__init__()* de la clase *Empleado*. El parámetro *self* de ese constructor queda automáticamente asignado con la dirección del objeto que se está creando (sin tener que enviar un parámetro actual desde el exterior para que sea asignado en *self*). El resto de los parámetros actuales enviados (los valores 1, 'Juan', 'Calle 1', 10000, y 10) son asignados en forma normal en los parámetros formales *leg*, *nom*, *direc*, *suel* y *ant* y con esos valores se crean los campos o atributos del objeto. El bloque de acciones del constructor tiene las instrucciones:

```
self.legajo = leg
self.nombre = nom
self.direccion = direc
self.sueldo = suel
self.antiguedad = ant
```

lo cual significa que para el registro u objeto *self* se está creando el campo self.legajo con el valor del parámetro formal *leg*, y algo similar para los campos o atributos self.nombre, self.direccion, self.sueldo y self.antiguedad. La dirección del registro u objeto self es la que

automáticamente retorna el constructor *__init__()*, y esa dirección se asigna en la variable *e1* que se mostró en el ejemplo.

Una vez que el registro u objeto ha sido creado e inicializado de esta forma, puede ser utilizado de allí en adelante en la misma forma en que hemos visto en esta Ficha: se usa el nombre de la variable registro seguido de un punto y luego el nombre del campo que se quiere acceder, como se ve en la función *write()* del ejemplo:

```
# una función para mostrar un objeto o instancia de la clase Empleado
def write(empleado):
    print("\nLegajo:", empleado.legajo, end=' ')
    print("- Nombre:", empleado.nombre, end=' ')
    print("- Direccion:", empleado.direccion, end=' ')
    print("- Sueldo:", empleado.sueldo, end=' ')
    print("- Antiguedad:", empleado.antiguedad, end=' ')
```

Para aplicar lo visto, veamos ahora un caso de estudio muy conocido, como es el de la representación de puntos en un plano. El siguiente enunciado formaliza la idea:

Problema 46.) *Desarrollar un programa que permita manejar puntos en un plano. Por cada punto se deben indicar sus coordenadas (que pueden ser números en coma flotante) y una cadena de caracteres a modo de descriptor del punto cuando se muestren sus valores en pantalla. Incluir un menú de opciones que permita:*

1. *Cargar por teclados los datos de un punto y mostrar esos datos en pantalla.*
2. *Cargar por teclado los datos de un punto, y mostrar la distancia al origen desde ese punto.*
3. *Cargar por teclado los datos de dos puntos, y mostrar la pendiente de la recta que los une.*

Discusión y solución: Un punto *p1* en un plano bidimensional gestionado a través de un par de ejes cartesianos, puede ser representado en forma simple a partir de sus dos coordenadas *(x, y)* que dan la distancia horizontal y vertical de ese punto *p1* al origen del sistema[1], como se ve en la *Figura 5* de *página 377*.

Esto permite rápidamente pensar en que un punto puede representarse en un programa en Python mediante un registro que contenga tres campos: dos valores de tipo *float* para las coordenadas, y una cadena de caracteres para el descriptor (que en el gráfico anterior es "p1"). Dentro del proyecto *[F18] Registros*, el módulo *geometry.py* contiene la declaración del registro o clase *Point* que usaremos para representar puntos, incluyendo su constructor:

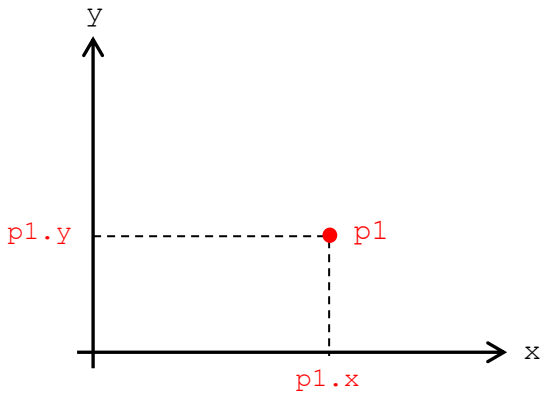
```
class Point:
    def __init__(self, cx, cy, desc='p'):
        self.x = cx
        self.y = cy
        self.descripcion = desc
```

El constructor *__init__(self, cx, cy, desc='p')* crea los campos del punto *self* que toma autmáticamente como parámetro. El parámetro formal *desc* tiene por defecto asignada la

[1] El uso de un sistema de ejes cartesianos para representar objetos o propiedades en función de dos o más coordenadas o valores aparece en las más insospechadas disciplinas, y en ese sentido el cine ha mostrado algunas escenas memorables. Un ejemplo concreto y muy recordado es la película *Dead Poets Society* (*La Sociedad de los Poetas Muertos*) de 1989, dirigida por *Peter Weir*. El profesor Keating (interpretado por *Robin Williams*) enseña literatura en un colegio secundario. El libro de texto que usan sus alumnos les explica cómo "medir" el valor de una poesía usando un sistema de ejes cartesianos… y el profesor Keating les ordena arrancar esa página del libro calificándola de basura… ¡Toda una perlita!

cadena 'p', por lo cual si ese parámetro se omite al invocar al constructor, el campo *descripcion* de *point* quedará valiendo 'p'.

Figura 5: Representación de un punto en un sistema cartesiano (primer cuadrante).



El mismo módulo *geometry.py* contiene el resto de la funciones pedidas en el enunciado. El cálculo de la distancia de un punto hasta el origen de coordenadas del sistema, se realiza con la función *distance()*, que simplemente aplica el Teorema de Pitágoras:

```
def distance(point):
    # Pitágoras...
    return math.sqrt(pow(point.x, 2) + pow(point.y, 2))
```

El módulo *geometry.py* provee una función *to_string()* simple pero muy cómoda, que crea una cadena de caracteres a partir de un registro u objeto *point* de tipo *Point*, y retorna esa cadena lista para ser mostrada en consola si fuese necesario:

```
def to_string(point):
    r = str(point.descripcion) + '(' + str(point.x) + ',' + str(point.y) + ')'
    return r
```

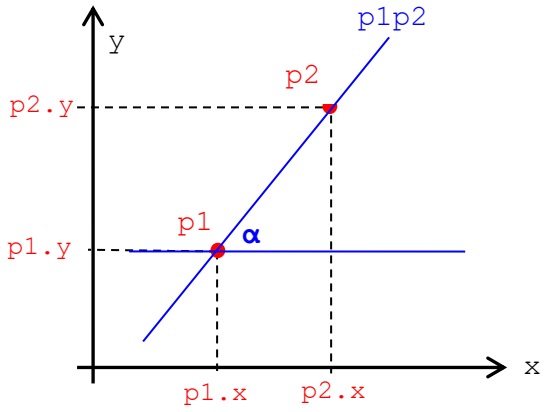
El cálculo de la pendiente de la recta que une a dos puntos *p1* y *p2*, se realiza mediante la función *gradient(p1, p2)*. Esa pendiente no es otra cosa que la *tangente trigonométrica* del ángulo *α* que forma la recta *p1p2* con la horizontal que pasa por *p1* (ver *Figura 6*, en *página 378*).

Sólo hay que tomar la precaución de controlar que *Δx* no sea cero (es decir, controlar que los puntos *p1* y *p2* no formen una recta vertical) pues en ese caso el cociente no puede calcularse (y se dice que la pendiente de una recta vertical es indefinida).

En base a lo expuesto, la función *gradient(p1, p2)* es la siguiente:

```
def gradient(p1, p2):
    # calcular "delta y" y "delta x"
    dy = p2.y - p1.y
    dx = p2.x - p1.x

    # si los puntos no forman una recta vertical,
    # retornar la pendiente...
    if dx != 0:
        return dy / dx

    # de otro modo, la pendiente es indefinida...
    return None
```

Figura 6: Representación geométrica de la pendiente de la recta *p1p2*.



pendiente(p1p1) = tg(α) = Δy / Δx
= (p2.y – p1.y) / (p2.x – p1.x)

Asumiendo que el módulo *geometry.py* existe y contiene estas declaraciones y funciones (ver proyecto *[F18] Registros*), entonces el programa completo es el siguiente (de nuevo, ver proyecto *[F18] Registros*, modelo *test06.py*):

```
__author__ = 'Cátedra de AED'


from geometry import *


def opcion1():
    cx = float(input('Coordenada x: '))
    cy = float(input('Coordenada y: '))
    p = Point(cx, cy)

    print(to_string(p))


def opcion2():
    cx = float(input('Coordenada x: '))
    cy = float(input('Coordenada y: '))
    p = Point(cx, cy)

    d = distance(p)
    print('Distancia al origen:', d)


def opcion3():
    cx1 = float(input('Punto 1 - Coordenada x: '))
    cy1 = float(input('Punto 1 - Coordenada y: '))
    p1 = Point(cx1, cy1)

    cx2 = float(input('Punto 2 - Coordenada x: '))
    cy2 = float(input('Punto 2 - Coordenada y: '))
    p2 = Point(cx2, cy2)

    pd = gradient(p1, p2)
    print('Pendiente de la recta que los une:', pd)


def menu():
```

```
    op = -1
    while op != 4:
        print('1. Cargar y mostrar un punto')
        print('2. Distancia al origen')
        print('3. Pendiente de la recta que une dos puntos')
        print('4. Salir')
        op = int(input('Ingrese opcion: '))

        if op == 1:
            opcion1()
        elif op == 2:
            opcion2()
        elif op == 3:
            opcion3()


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

4.] Arreglos de registros (o vectores de registros) en Python.

En muchas aplicaciones será más útil almacenar registros completos en cada casillero de un arreglo unidimensional, en lugar de usar arreglos paralelos como mostramos en la sección 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 sólo 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 también suele designarse como un *vector de registros*.

Supongamos que se tiene definido un registro *Estudiante* y que se desea almacenar varios registros de ese tipo en un arreglo. Suponga que se cuenta con el tipo *Estudiante* ya definido y con una función constructora *__init()__* para crear y asignar los campos a cada registro de ese tipo:

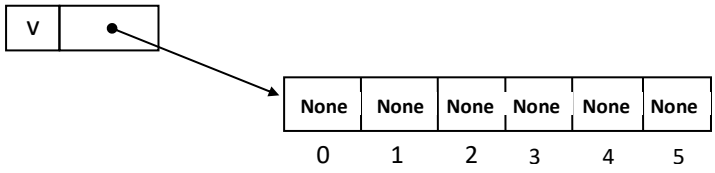
```
class Estudiante:
    def __init__(self, leg=0, nom='', pro=0):
        self.legajo = leg
        self.nombre = nom
        self.promedio = pro
```

Si se conoce de antemano la cantidad *n* de registros 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 registro de tipo *Estudiante*, aunque todavía no se han creado esos registros. El aspecto que tendría este arreglo en este momento en memoria sería como el que se muestra a continuación:

Figura 7: 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 registros 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 registro 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 registro, 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 registros. Una mejor forma de proceder, podría ser que no sólo se creen los registros vacíos, sino que también se proceda a cargar por teclado los datos a almacenar en cada uno, y *luego se creen los registros con sus campos 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)
```

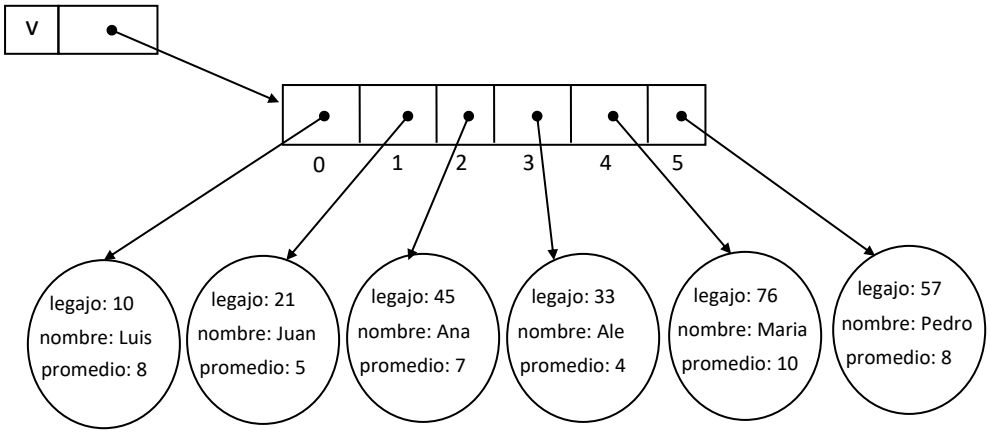
Está claro que puede lograrse el mismo resultado dejando los valores default de la función constructora *__init()__*, y luego *creando y asignando directamente los campos en cada casillero*. Lo anterior es equivalente a:

```
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 registro 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 campos.

Las dos secuencias anteriores crean diferentes registros y los asignan en distintas casillas del arreglo, que podría verse ahora como sigue (suponga que los datos cargados por teclado fueron efectivamente los que se ven en la figura):

Figura 8: Un arreglo con *n = 6* referencias a registros ya creados de tipo *Estudiante*.



Sólo cuando cada registro 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 registro en la casilla *i*, y por lo tanto algo como *v[i].legajo* accede al número de legajo almacenado en el registro *v[i]*. Pero esto sólo es válido si *v[i]* es efectivamente un registro de tipo *Estudiante*: si se intenta acceder a un campo desde una referencia que no apunta a un registro de ese tipo, el programa se interrumpirá lanzando un mensaje de error por campo inexistente.

Finamente, el arreglo se usa como se usaría a cualquier arreglo: si se sabe qué clase de registros 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 *test07.py* (incluido en el proyecto *[F18] Registros* que viene con esta Ficha), y es el siguiente:

```
__author__ = 'Cátedra de AED'


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


def write(est):
    print('Legajo:', est.legajo, end=' ')
    print('- Nombre:', est.nombre, end=' ')
    print('- Promedio:', est.promedio)


def validate(inf):
    n = inf
    while n <= inf:
        n = int(input('Cantidad de elementos (mayor a ' + str(inf) + '): '))
```

```
        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:
            write(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 mínimos para poder hacer el curso: '))
    print()

    # ordenar alfabéticamente el arreglo...
    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 registros de tipo *Estudiante* (tipo que fue definido 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 registros de tipo *Estudiante* en cada componente (cosa que se 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[i].legajo:
                estudiantes[i], estudiantes[j] = estudiantes[j], estudiantes[i]
```

Puede verse que en la condición se accede al campo *legajo* de cada registro, y se comparan sus valores: ahora cada casillero contiene un registro 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 registros, *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 registros, 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 *[F18] Registros* que acompaña a esta Ficha. Dentro de ese proyecto, el módulo *insumos.py* define la clase *Insumo* para manejar registros que representen insumos. La clase incluye su función constructora *__init__()* con un parámetro por cada campo previsto en el registro, y una función adicional *to_string()* que retorna una cadena con el contenido del registro tomado como parámetro, de forma que esa cadena ya tiene el formato adecuado para ser visualizada en pantalla si fuese el caso:

```
__author__ = 'Cátedra de AED'


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

La función *to_string()* usa en forma directa los campos que sean de tipo cadena y también convierte cada campo numérico del registro 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(insumo.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 sólo 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 nuestra función *to_string()* original:

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

Cada campo del registro *insumo* que se toma como parámetro 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 el registro fuesen:

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

entonces el siguiente script:

```
print(to_string(insumo))
```

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 *test08.py* del mismo proyecto, y lo mostramos a continuación:

```
import insumos

__author__ = 'Cátedra de AED'


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(insumos.to_string(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(insumos.to_string(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(insumos.to_string(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(insumos.to_string(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

__author__ = 'Cátedra de AED'
```

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. Sólo 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 registro con nuestra ya analizada función *to_string()* (tomada desde el módulo *insumos.py*):

```
def display_all(pieza):
    for insumo in pieza:
        print(insumos.to_string(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(insumos.to_string(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 registro analizado en la vuelta actual tiene su campo 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 registros 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 a nuestra función *insumos.to_string()*:

```
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(insumos.to_string(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 la función *insumos.to_string()*:

```
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(insumos.to_string(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 *test09.py* (incluido en el proyecto *[F18] Registros* que viene con esta Ficha). Incluye el módulo *socios.py* con la definición de la clase *Socio*, su función constructora *__init()__* y la ya típica función *to_string()*:

```
__author__ = 'Cátedra de AED'


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

El programa completo para resolver el problema está incluido en el archivo *test09.py* del mismo proyecto, y se transcribe a continuación:

```
import socios

__author__ = 'Cátedra de AED'


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(socios.to_string(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(socios.to_string(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(socios.to_string(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 en para el problema anterior, por lo que confiamos en que el alumno 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.

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 campo *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 campo *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:

```
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.

5.] Matrices de registros en Python.

Del mismo modo que lo hecho para arreglos unidimensionales, se pueden crear arreglos bidimensionales que contengan referencias a registros si fuese necesario. La idea es esencialmente la misma que para crear una matriz de valores simples. Supongamos que el tipo registro *Estudiante* está ya definido y listo para usar en base al esquema que sigue:

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

Suponga también que se quiere crear una matriz est en la que cada fila *f* contenga los datos de los estudiantes que cursan en el año o nivel *f*, y cada columna *c* se usa para distinguir el número de orden de cada estudiante en ese nivel. Entonces la creación de esa matriz *est* de

fils filas y *cols* columnas en la que cada casilla contenga una referencia a un registro de tipo *Estudiante* puede comenzar definiendo la matriz mediante *creación por comprensión*, en la forma siguiente [1]:

```
est = [[None] * cols for f in range(fils)]
```

La instrucción anterior crea una variable *est* de tipo *list*, que contendrá tantos elementos (a modo de *filas*) como indique la variable *fils*. Cada uno de esos elementos será a su vez una lista con *cols* elementos (a modo de columnas) valiendo *None* (relea la *Sección* ***¡Error! No se encuentra el origen de la referencia.***, *página* ***¡Error! Marcador no definido.*** y siguientes en esta misma Ficha).

Como cada casillero de la matriz en este momento vale *None* y no hay realmente ningún registro de tipo *Estudiante*, lo siguiente *es crear esos registros y asignarlos en cada casilla*. Un esquema de ciclos anidados como el que se muestra en la siguiente función permite lograrlo, realizando también la carga por teclado de los datos de cada registro:

```
def read(fils, cols):
    est = [[None] * cols for f in range(fils)]
    print('Ingrese los datos de cada estudiante...')
    for f in range(fils):
        print('Nivel', f, ':')
        for c in range(cols):
            print('\tEstudiante número', c)
            leg = int(input('\t\tLegajo: '))
            nom = input('\t\tNombre: ')
            pro = float(input('\t\tPromedio: '))

            est[f][c] = Estudiante(leg, nom, pro)
            print()
    return est
```

Y el resto es simple cuestión de extrapolar las técnicas de recorrido ya conocidas para procesar el contenido de la matriz, recordando que ahora cada casillero contiene un registro. Por lo tanto, y a modo de ejemplo, el acceso al campo *legajo* del registro ubicado en la casilla *est[f][c]* se realiza con el identificador *est[f][c].legajo* y luego asignando en ese campo un valor o usando su valor como parte de una expresión cualquiera. El siguiente problema o caso de análisis permite ilustrar todo lo visto, aclarar las ideas y poner en práctica los mecanismos explicados:

Problema 50.) *El responsable del consorcio de un pequeño barrio cerrado desea obtener una estadística de los gastos incurridos en el barrio a lo largo de un trimestre por las* ***n*** *propiedades que hay en el barrio. Para ello, se solicita crear tipo de registro designado como* ***Consumo****, con los campos que describan el consumo realizado en un mes dado por una propiedad: en uno de los campos almacenar el importe pagado por electricidad en ese mes, en otro el importe gastado en gas, y en otro el importe gastado en servicio medido de agua. Se debe crear una matriz* ***cons*** *de referencias a registros de tipo Consumo, en la que cada columna represente uno de los tres meses del trimestre analizado (numerados de 0 a 2) y en la que cada fila represente una de las* ***n*** *propiedades (numeradas de 0 a n-1). El registro de tipo Consumo almacenado en la casilla* ***cons[i][j]****, representará entonces los gastos realizados por la propiedad i en el mes j. Se pide un que permita:*

a.) *Cargar los datos de la matriz.*

b.) *Mostrar adecuadamente los datos de toda la matriz.*

c.) *Mostrar por pantalla un listado general en el cual se indique el gasto acumulado por cada propiedad en el trimestre (es decir, una totalización por filas de la matriz).*

d.) *Mostrar por pantalla un listado general en el cual se indique el gasto acumulado por todas las propiedades en cada mes del trimestre (es decir, una totalización por columnas de la matriz).*

Discusión y solución: El proyecto *F[19] Registros* que acompaña a esta Ficha contiene un modelo *test10.py* con el programa completo que resuelve este problema.

```
__author__ = 'Cátedra de AED'


class Consumo:
    def __init__(self, el, gs, ag):
        self.electricidad = el
        self.gas = gs
        self.agua = ag


def to_string(cons):
    r = ''
    r += '{:<35}'.format('Gasto de electricidad: ' + str(cons.electricidad))
    r += '{:<17}'.format('Gas: ' + str(cons.gas))
    r += '{:<17}'.format('Agua: ' + str(cons.agua))
    return r


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 read(fils, cols):
    cons = [[None] * cols for f in range(fils)]
    print('Ingrese los montos consumidos por cada rubro...')
    for f in range(fils):
        print('Propiedad', f, ':')
        for c in range(cols):
            print('\tMes', c)
            el = float(input('\t\tGasto en electricidad: '))
            gs = float(input('\t\tGasto en gas: '))
            ag = float(input('\t\tGasto en agua: '))
            cons[f][c] = Consumo(el, gs, ag)
            print()
    return cons


def display(cons):
    filas, columnas = len(cons), len(cons[0])
    print('Planilla de gastos mensuales por propiedad...')
    for f in range(filas):
        print('Propiedad', f)
        for c in range(columnas):
            print(to_string(cons[f][c]))
    print()


def total_per_property(cons):
    filas, columnas = len(cons), len(cons[0])
```

```
    print('Gastos por propiedad en cada trimestre')
    for f in range(filas):
        ac = 0
        for c in range(columnas):
            t = cons[f][c].electricidad + cons[f][c].gas + cons[f][c].agua
            ac += t
        print('Propiedad:', f, '\tGasto total:', ac)


def total_per_month(cons):
    filas, columnas = len(cons), len(cons[0])
    print('Gastos por mes entre todas las propiedades')
    for c in range(columnas):
        ac = 0
        for f in range(filas):
            t = cons[f][c].electricidad + cons[f][c].gas + cons[f][c].agua
            ac += t
        print('Mes:', c, '\tGasto total:', ac)


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

    # crear y cargar la matriz de consumos...
    cols = 3
    cons = read(fils, cols)
    print()

    # mostrar todos los datos cargados...
    display(cons)
    print()

    # acumulación por filas...
    total_per_property(cons)
    print()

    # acumulación por columnas...
    total_per_month(cons)


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

Anexo: Temas Avanzados

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

a.) Generación de un arreglo de registros 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 registros 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 registros 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 campo de los registros 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 51.) *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 *[F18] Registros* que acompaña a esta Ficha. Como de costumbre, la clase *Vehiculo*, su función *__init__()* y la función *to_string()* se incorporan en un módulo aparte, que se llama *vehiculos.py* en el mismo proyecto:

```
__author__ = 'Cátedra de AED'


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


def to_string(vehiculo):
```

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

El programa completo está desarrollado en el archivo *test11.py* del mismo proyecto, y lo mostramos a continuación:

```
import random
import vehiculos

__author__ = 'Cátedra de AED'


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=0, mx=14):
    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(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 precede 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(vehiculos.to_string(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(vehiculos.to_string(v))

    if not exists:
        print('No hay 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(vehiculos.to_string(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 veiculos 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 del registro pedido 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 registros.

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 campos 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 campo 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 patentes:

```
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.

Créditos

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

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

Bibliografía

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

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

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