



Taller 9

Fecha: Junio de 2021

Profesor: Fray León Osorio Rivera

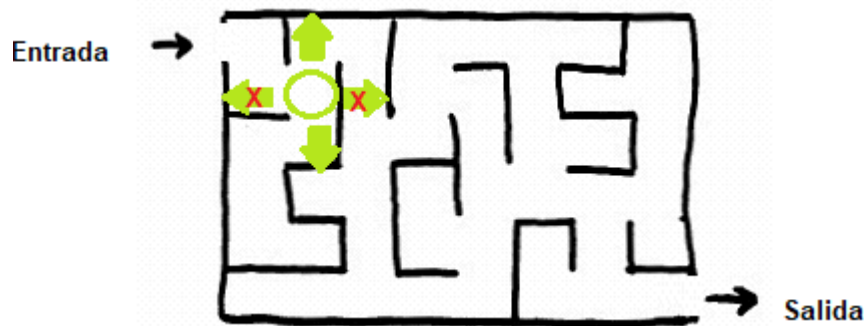
Competencia a evaluar: Aplicar los conceptos básicos de la orientación a objetos y las estructuras de datos en el desarrollo de una aplicación con interfaz gráfica de usuario.

NOTAS:

- Este taller se debe hacer como preparación para examen o prácticas. En ningún caso representará una calificación.
- El primer ejercicio se entrega resuelto como ejemplo para el desarrollo de los demás.

Elaborar el diagrama de clases básico y la respectiva aplicación en el lenguaje *Python* para los siguientes enunciados:

1. Para la solución de un laberinto se requieren evaluar varias alternativas cada vez que avanza una posición en él:
 - Puedo desplazarme sólo 4 sentidos (nunca en diagonal)
 - Hay muros que no se pueden atravesar
 - Se debe evitar devolverse para no volver a repetir ubicaciones.
 - Puede que no se pueda encontrar la salida



Elaborar una aplicación que, dado el laberinto con la entrada y salida, permita indicar si existe una ruta desde la entrada hasta la salida.

R/

Para comprender este ejercicio, es importante tener en cuenta los siguientes fundamentos:

Matrices

Una estructura muy útil para manejar volúmenes de datos son los arreglos bidimensionales que pueden ser vistos como tablas de valores. Cada elemento de un arreglo bidimensional está simultáneamente en una fila y en una columna. En matemáticas, a los arreglos bidimensionales se les llama matrices, y son muy utilizados en problemas de Ingeniería.



	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	1	0	0	1	0	0	0	1	0	1
3	1	0	0	1	0	0	0	1	0	1
4	1	0	0	0	0	1	1	0	0	1
5	1	0	1	1	1	0	0	0	0	1
6	1	0	0	0	1	0	0	0	0	1
7	1	0	1	0	0	0	1	0	0	1
8	1	0	1	1	1	0	1	1	0	1
9	1	1	0	0	0	0	0	0	0	1
10	1	1	1	1	1	1	1	1	1	1

Por ejemplo, la anterior matriz es la que se utilizará para representar el laberinto, donde los 1 son los muros y los 0 son los espacios para desplazarse. Su forma de definirse en Python sería:

```
laberinto = [[1,1,1,1,1,1,1,1,1,1], \  
             [1,0,0,1,0,0,0,1,0,1], \  
             [1,0,0,1,0,0,0,1,0,1], \  
             [1,0,0,0,0,1,1,0,0,1], \  
             [1,0,1,1,1,0,0,0,0,1], \  
             [1,0,0,0,1,0,0,0,0,1], \  
             [1,0,1,0,0,0,1,0,0,1], \  
             [1,0,1,1,1,0,1,1,0,1], \  
             [1,1,0,0,0,0,0,0,0,1], \  
             [1,1,1,1,1,1,1,1,1,1]]
```

Donde puede observarse que se trata de un vector de vectores.

Para acceder a cada valor se utilizarían dos índices:

```
valor = laberinto [2][5]
```

la anterior instrucción asignaría 0 en la variable *valor*, ya que esto es lo almacenado en la matriz *laberinto* en dicha posición. Se hace la salvedad que en los programas en Python la primera posición no es 1 sino 0.

Como se trata de un arreglo bidimensional, tiene 2 dimensiones:

```
filas = len(laberinto)  
columnas = len(laberinto[0])
```

El total de filas que se obtiene con la función *len()* directamente a la matriz y el total de columnas que se obtiene preguntando la longitud del arreglo en la posición 0.

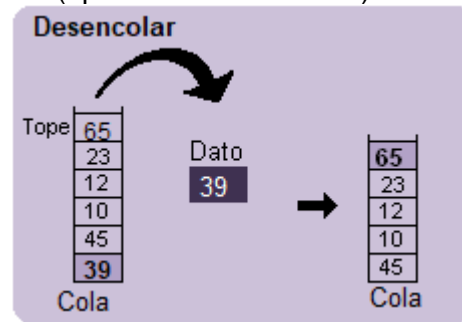
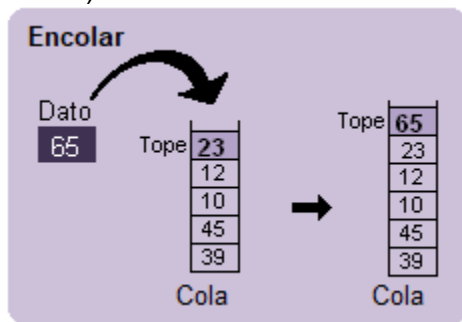


Concepto de Cola

Una **cola** es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción se realiza por un extremo y la operación de extracción se realiza por el otro. Es una estructura de tipo *FIFO* (del inglés *First In First Out*), debido a que el primer elemento que ingresa será también el primero en salir.

Son dos los cambios básicos que se pueden hacer en una cola:

- Agregar un nuevo elemento (operación **Encolar**)
- Quitar el primer elemento agregado (operación **Desencolar**)




Para realizar estas operaciones existen algunas restricciones:

- No se puede *desencolar* si no hay elementos en la cola (condición **Cola Vacía**).

ColaVacía

Tope
Cola

La siguiente es la descripción de la clase *Cola*:

Cola
 datos
encolar() desencolar(): objeto vacía(): booleano

- El método *encolar()* permite acolar un dato en la cola.
- El método *desencolar()* permite desacolar un objeto de la cola
- El método *vacía()* devuelve un valor booleano que indica si la cola está vacía

El siguiente sería el respectivo código en *Python*:

```
class Cola:
    #Metodo constructor
    def __init__(varClase):
        varClase.datos=[]

    def encolar(varClase, dato):
```



```
varClase.datos.append(dato)

def desencolar(varClase):
    if not varClase.vacia():
        dato=varClase.datos[0]
        del varClase.datos[0]
        return dato
    else:
        return None

def vacia(varClase):
    return len(varClase.datos)==0
```

Se tendrá una clase que se denominará *Punto* para almacenar las coordenadas de una ubicación en particular del laberinto. Tendrá funcionalidad para comparar si 2 puntos son iguales.

El siguiente es el código:

```
class Punto():

    def __init__(varClase, x, y):
        varClase.x = x
        varClase.y = y

    def esIgual(varClase, punto):
        return varClase.x == punto.x and varClase.y == punto.y
```

La clase que implementa la funcionalidad del laberinto es la siguiente:

```
class Laberinto():

    def __init__(varClase, laberinto, \
                xEntrada, yEntrada, \
                xSalida, ySalida):
        varClase.entrada = Punto(xEntrada, yEntrada)
        varClase.salida = Punto(xSalida, ySalida)
        varClase.laberinto = laberinto

        varClase.filas = len(laberinto)
        varClase.columnas = len(laberinto[0])

    def mostrar(varClase):
        for i in range(0, varClase.filas):
            linea=""
            for j in range(0, varClase.columnas):
                if varClase.laberinto[i][j] == 1:
                    linea += "|"
                elif varClase.laberinto[i][j] == 0:
                    linea += " "
```



```
elif varClase.laberinto[i][j] == -1:
    linea += "-"
else:
    linea += str(varClase.laberinto[i][j])
print(linea)

def asignar(varClase, punto, valor):
    varClase.laberinto[punto.x][punto.y] = valor

def estaLibre(varClase, unPunto):
    return varClase.laberinto[unPunto.x][unPunto.y] == 0

def resolver(varClase):
    cola = Cola()
    varClase.asignar(varClase.salida, 0) # marcar el punto de salida

    punto = varClase.entrada
    cola.encolar(punto)
    varClase.asignar(punto, -1) # Asignarlo a -1 para evitar retroceder y
repetir la búsqueda
    while not cola.vacia():
        punto = cola.desencolar()
        if punto.esIgual(varClase.salida): # Salida encontrada, ruta de
salida
            return True # Devuelve true cuando se encuentra una ruta
        for di in range(0, 4): # escanea cíclicamente cada posición
            heAvanzado = False
            if di == 0 and punto.x>0:
                #avanzar a la IZQUIERDA
                heAvanzado = True
                puntoInteres=Punto(punto.x-1, punto.y)
            elif di ==1 and punto.y<varClase.columnas-1:
                #avanzar hacia ABAJO
                heAvanzado = True
                puntoInteres=Punto(punto.x, punto.y+1)
            elif di ==2 and punto.x<varClase.filas-1:
                #avanzar a la DERECHA
                heAvanzado = True
                puntoInteres=Punto(punto.x+1, punto.y)
            elif di ==3 and punto.y>0:
                #avanzar hacia ARRIBA
                heAvanzado = True
                puntoInteres=Punto(punto.x, punto.y-1)

            if heAvanzado:
                if varClase.estaLibre(puntoInteres):
                    cola.encolar(puntoInteres)
                    varClase.asignar(puntoInteres, -1)

    return False
```

En ella se hace aprovechamiento de la clase *Cola* para poder almacenar en cada ubicación los puntos hacia dónde puedo desplazarme para poder evaluar cada alternativa.



Siempre que haya un punto encolado hay alternativas. Si llegase a estar la cola de puntos vacía, el laberinto no tiene solución.

Se tienen un método para hacer visible el laberinto en cualquier instante:

Ahora bien, el código que resuelve el problema sería el siguiente:

```
from Laberinto import Laberinto

laberinto = [[1,1,1,1,1,1,1,1,1,1], \
              [1,0,0,1,0,0,0,1,0,1], \
              [1,0,0,1,0,0,0,1,0,1], \
              [1,0,0,0,0,1,1,0,0,1], \
              [1,0,1,1,1,0,0,0,0,1], \
              [1,0,0,0,1,0,0,0,0,1], \
              [1,0,1,0,0,0,1,0,0,1], \
              [1,0,1,1,1,0,1,1,0,1], \
              [1,1,0,0,0,0,0,0,0,1], \
              [1,1,1,1,1,1,1,1,1,1]]

l = Laberinto(laberinto, 1, 0, len(laberinto)-1, len(laberinto[0])-2)

print("El laberinto es:")
l.mostrar()

if l.resolver():
    print("El laberinto resuelto es:")
    l.mostrar()

else:
    print("El laberinto no se puede resolver")
```

Cuya ejecución luciría así cuando encuentra que el laberinto tiene solución:



El laberinto es:

```
| | | | | | | | | |
|   |   |   |   |
|   |   |   |   |
|   |   | |   |
|   | | |   |
|   |   |   |
|   |   |   |
|   |   |   |
|   | | | | |
|   |   |   |
| | | | | | | | | |
```

El laberinto resuelto es:

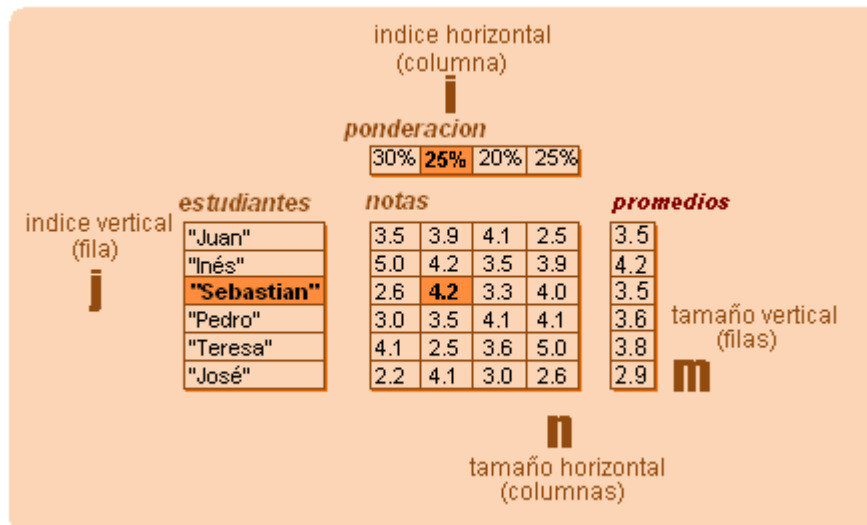
```
| | | | | | | | | |
*** | *** |   |
| ** | *** | * | | | |
| **** | | ** |
| * | | **** |
| *** | **** |
| * | *** | ** |
| * | | * | | * |
| | **** |
| | | | | | * |
>>>
```

2. Para el registro de las calificaciones de un curso universitario, se debe permitir digitar:
- Los nombres de los estudiantes.
 - Los valores porcentuales (ponderación) de cada una de las calificaciones.
 - Cada una de las calificaciones obtenidas por los estudiantes.

Se pide obtener a partir de los anteriores datos, el estudiante con mayor promedio y el promedio general del curso.

R/

Las estructuras de datos necesarias para solucionar este problema se pueden ver en la siguiente gráfica:



Se trata de 3 vectores y una matriz.

- **Datos de Entrada:**

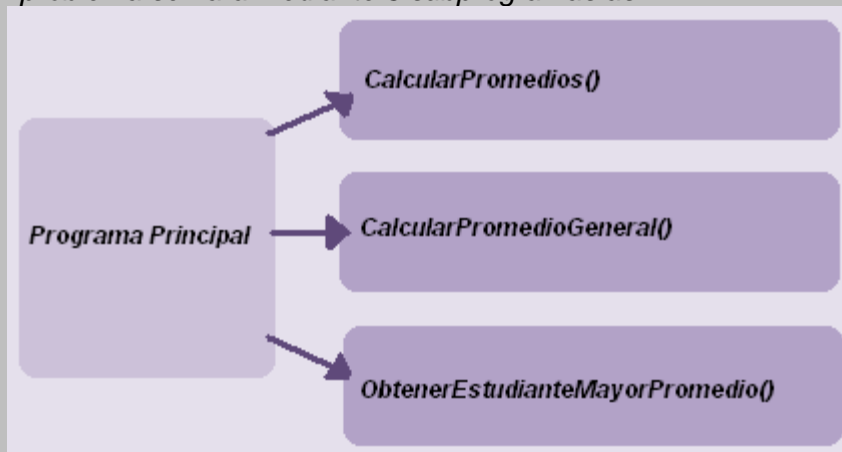
- Total de estudiantes (*m*)
- Total de porcentajes (*n*)
- Lista de estudiantes (*estudiantes*) - Vector
- Lista de ponderaciones (*ponderacion*) - Vector
- Lista de calificaciones (*notas*) - Matriz

- **Datos de Salida:**

- Estudiante mayor promedio (*emp*)
- Promedio general curso (*pg*)

- **Proceso:**

La solución del problema se hará mediante 3 subprogramas así:



- Procedimiento **CalcularPromedios()** que se encarga de calcular el promedio que obtenga cada estudiante a partir de sus calificaciones. Tiene 4 parámetros de entrada: las calificaciones, los porcentajes, el total de estudiantes y, el total de porcentajes. Y 1 parámetro de salida: Los promedios calculados.



- Función **CalcularPromedioGeneral** que basado en el anterior calculo, obtiene el promedio de todos los promedios de los estudiantes. Tiene 2 parámetros de entrada: Los promedios calculados y el total de estudiantes.
- Función **ObtenerEstudianteMayorPromedio** que basado en los promedios de los estudiantes, obtiene el nombre del estudiante con el mayor de estos promedios. Tiene 3 parámetros de entrada: Los promedios calculados, los nombres de los estudiantes y el total de estudiantes.

Algoritmo

Programa principal:

```

Lea m
Lea n
Para i ← 0 Hasta m-1
    Lea estudiantes[i]
    Para j ← 0 Hasta n
        Lea notas[i][j]
    FinPara
FinPara
Para j ← 0 Hasta n-1
    Lea ponderacion [j]
FinPara
CalcularPromedios( notas, ponderacion, m, n, promedios)
pg ← CalcularPromedioGeneral( promedios, m)
Imprima pg
emp ← ObtenerEstudianteMayorPromedio( promedios, estudiantes, m )
Imprima emp
Fin
    
```

Subprogramas:

```

Procedimiento CalcularPromedios( notas, ponderacion, tEstudiantes, tNotas,
promedios)
Para i ← 0 Hasta tEstudiantes-1
    promedios[i] ← 0
    Para j ← 0 Hasta tNotas-1
        promedios[i] ← promedios[i] + notas[i][j] * ponderaciones[j] /
100
    FinPara
FinPara
FinProcedimiento
    
```

```

Funcion CalcularPromedioGeneral( promedios, tEstudiantes)
suma ← 0
Para i ← 0 Hasta tEstudiantes-1
    suma ← suma + promedios[i]
FinPara
Retornar suma / tEstudiantes
    
```

```

Funcion ObtenerEstudianteMayorPromedio( promedios, estudiantes, tEstudiantes
)
pemp ← 0
Para i ← 1 Hasta tEstudiantes-1
    
```



```
    Si promedios[i] > promedios[pemp] Entonces  
        pemp ← i  
    FinSi  
FinPara  
Retornar estudiantes[i]
```