Primer laboratorio guiado sobre Python

Objetivos

1. Aprender a crear una clase y funciones controladoras en Python.

Procedimiento

1. Abra NotePad++ o el IDE que prefiera. Se recomienda NotePad++ por ser muy liviano y sencillo.

2. Vamos a construir una clase que representará un grafo con pesos en las aristas. Para esto se usará una matriz de adyacencias como la estructura de datos. Los vértices del grafo estarán identificados con números entre cero y n - 1, donde n es la cantidad total de vértices en el grafo. Se ocupará entonces una matriz n x n de valores enteros positivos. Los pesos en las aristas deben ser valores positivos. Para indicar que no existe una arista entre dos vértices i,j las correspondientes entradas de la matriz ([i,j] y [j,i]) guardarán el valor "float('inf')" que en Python representa el máximo valor posible. Indicará que el peso entre el vértice i al vértice j (o viceversa) es infinito. A continuación SE RECOMIENDA escribir el código NO COPIAR Y PEGAR.

3. Para construir esta clase que denominaremos GrafoPesado se requerirán dos componentes:

```
import numpy as np
import random
```

4. Para definir la nueva clase se agrega al archivo:

```
class GrafoPesado:
      """Representa una red compleja de vertices (vrt) y adyacencias (ady).
      SUPUESTOS:
      1. los vertices se identifican con enteros entre 0 y n
      2. las aristas tienen pesos mayores o iguales a cero. """
```

Observe que:

- La sangría es obligatoria, en NotePad++ corresponde a una marca de tabulación "TAB".
- Los dos puntos al final del nombre de la clase son obligatorios.
- La triple comilla se puede usar para comentarios

5. **En adelante se agrega una sangría antes de cada "def"**. Se agrega el único constructor que puede tener una clase en Python:

```
#EFE: Construye un GrafoPesado vacio de n x n.
# El grafo queda inicializado con pesos máximos float('inf') que indica que no hay aristas.
def __init__(self, n):
       self.n = n                          # se guarda la cantidad de vertices
       self.matriz_adys = np.empty( (n, n) )    # representa la matriz de adyacencias
       self.matriz_adys[:] = float('inf') # se inicializan todas las arista en float('inf')
       return
```

Observe que:

- "__init__" es el nombre obligatorio del único constructor que puede tener cualquier clase.
- "self" representa el objeto que está siendo construido. En los demás métodos, representa al objeto que recibe la invocación del método. Es como *this en C++ o this en JAVA
- Los demás parámetros se separan por coma y que la definición de tipos para los parámetros no es obligatoria.
- La definición se variables de estado se hace por ejemplo con self.n = n. La variable de estado o variable de instancia queda automáticamente definida e inicializada. Definir e inicializar variables de estado o instancia es precisamente la función del constructor en cualquier clase de Python.
- La palabra clave "return" al final no es obligatoria, pero sí es una buena práctica para marcar con claridad el final del código de un método o función en general.

6. Se define un inicializador al azar:

```
# REQ: 0 < p < 1.
# EFE: Inicializa un GrafoPesado previamente creado con n vertices en el que el
# conjunto de adyacencias se determina aleatoriamente utilizando p como
# la probabilidad de que exista una adyacencia entre cualesquiera
# dos vertices. El peso de cada arista se escoge al azar uniformemente
# entre 0 y max-1.
def init__random(self, p: float, max: int):
	for i in range(self.n):
		for j in range(i+1,self.n):
			if (random.random() < p):# se genera un valor aleatorio entre 0 y 1
				peso = random.randint(0,max-1)
				self.matriz_adys[i][j] = peso
				self.matriz_adys[j][i] = peso
	return
```

Observe que:

- Los parámetros pueden incluir una definición de tipo, pero no es obligatoria y se hace a la inversa de Java y C++: primero aparece el nombre del parámetro y luego el tipo.
- "for i in range(...)" corresponde con un ciclo for.
- Para referenciar las variables de instancia o estado se debe usar "self".
- El mismo peso se guarda en las posiciones [i][j] y [j][i] porque se trata de un grafo y por tanto si existe la arista [i][j] debe existir [j][i] y tener exactamente el mismo peso.

7. Se agregan dos observadores booleanos básicos:

```
# EFE: retorna true si 0 <= idvrt < n.
# NOTA: idvrt significa "identificador de vertice".
def existe_vrt(self, idvrt: int):
	return 0 <= idvrt and idvrt < n

# EFE: retorna true si existe adyacencia entre los vertices idvrt_o e idvrt_d.
def existe_ady(self, idvrt_o: int, idvrt_d: int):
	return existe_vrt(idvrt_o) and exist_vrt(idvrt_d) and (self.matriz_adys[i][j] != float('inf'))
```

8. Se agrega un observador que permite obtener la lista de vértices adyacentes a un vértices dado. Es decir, la lista de aristas que involucran al vertice dado como origen y a todos los demás vértices que mantienen una arista con el vértice origen cuyo costo no sea infinito.

```
    # REQ: 0 <= idvrt < n.
    # EFE: retorna los idvrts de todos los vertices adyacentes a idVrt.
    def obt_idvrt_adys(self, idvrt: int):
           rsl = []
           for i in range (self.n):
                  if (self.matriz_adys[idvrt][i] != float('inf')):
                         rsl.append(i)
           return rsl
```

Observe que:

- rsl = [] define una lista local al método.
- rsl.append(...) permite agregar un valor a la lista previamente definida.

9. En Python se pueden sobrecargar operadores!!!! En el siguiente url aparecen todos los operadores que se pueden sobrecargar:
https://www.geeksforgeeks.org/operator-overloading-in-python/

A continuación se sobrecarga el operador [a,b,...] que permite obtener el valor del peso de una arista por medio de los identificadores de sus dos vértices:

```
    # REQ: 0 <= idvrt < n.
    # EFE: retorna el peso de la arista (idvrt_o, idvrt_d).
    # NOTA: implementa el operador [a,b,...].
    def __getitem__(self, indices):
           idvrt_o, idvrt_d = indices
           return self.matriz_adys[idvrt_o][idvrt_d]
```

10. Se agregan otros dos observadores básicos u "obtenedores":

```
    # EFE: retorna el total de vertices en el GrafoPesado.
    def obt_total_vrts(self):
           return self.n

    # EFE: retorna el total de adyacencias en el GrafoPesado.
    def obt_total_adys(self):
           suma_adys = 0
           for i in range(self.n):
                  for j in range(i+1,self.n):
                         if (self.matriz_adys[i][j] != float('inf')):
                                suma_adys += 1
           return suma_adys
```

11. Sólo para efectos de verificar que la construcción de la matriz de adyacencias es correcta, se agrega este observador que retorna una hilera que representa a la matriz para que sea impresa por consola por el invocador:

```
	def ver_matriz(self):
		return np.array2string(self.matriz_adys)
```

Observe que:

- La función np.array2string(...) genera una hilera legible para una matriz. Puede consultar la documentación correspondiente en:

https://docs.scipy.org/doc/numpy/reference/generated/numpy.array2string.html

12. En un archivo aparte, agregue una función que juegue el papel de controlador para hacer algunas pruebas a la clase recién creada:

```
import random
from grafo_pesado import GrafoPesado

def main():
	# genera semilla para numeros aleatorios,
	# OJO: no se puede invocar en el constructor porque repite los valores.
	random.seed()

	# construye un grafo con n == 10
	mi_grafo = GrafoPesado(10)

	# inicializa al azar el grafo previamente construido con n == 10
	mi_grafo.init__random(0.5,10)

	# se verifica la matriz de adyacencia interna
	print(mi_grafo.ver_matriz())

	# se despliegan los vértices adyacentes a cada vértice (0 <= peso) and (peso < float('inf'))
	for i in range(10):
		print(mi_grafo.obt_idvrt_adys(i))

	# se obtienen los pesos de cada arista y se imprimen siempre que existan
	for i in range(10):
		for j in range(10):
			if (mi_grafo[i,j] < float('inf')):
			print(mi_grafo[i,j])
	print("Total de vertices: ", mi_grafo.obt_total_vrts())
	print("Total de adyacencias: ", mi_grafo.obt_total_adys())

	return

main() # invoca la función main()
```

Observe que:

- Se importa del archivo de la clase la clase GrafoPesado. Un archivo puede contener más de una clase.
- Se inicializa el aleatorizador básico del componente "random".
- Primero se crea una instancia de GrafoPesado con 10 vértices y luego se inicializa.
- Se termina la función main() con la palabra clave "return" lo que es una buena práctica para mayor claridad del código, pero no es obligatoria.
- Al final se invoca la función "main()". El nombre de la función es puramente convencional, refleja la práctica común en C++, pero en Python, una función controladora puede tener cualquier nombre. Se recomienda en general asignar nombres significativos.

13. A continuación para ejecutar el programa:

13.1 Crear un ambiente de desarrollo para lo cual se abre una sesión de consola y se ejecuta la siguiente orden:

```
> python -m venv {nombre del ambiente virtual SIN blancos}
```

13.2 Activar el ambiente:

```
> .\{nombre del ambiente virtual SIN blancos}\Scripts\activate
```

Al final podrá desactivar el ambiente mediante:

```
> .\{nombre del ambiente virtual SIN blancos}\Scripts\deactivate
```

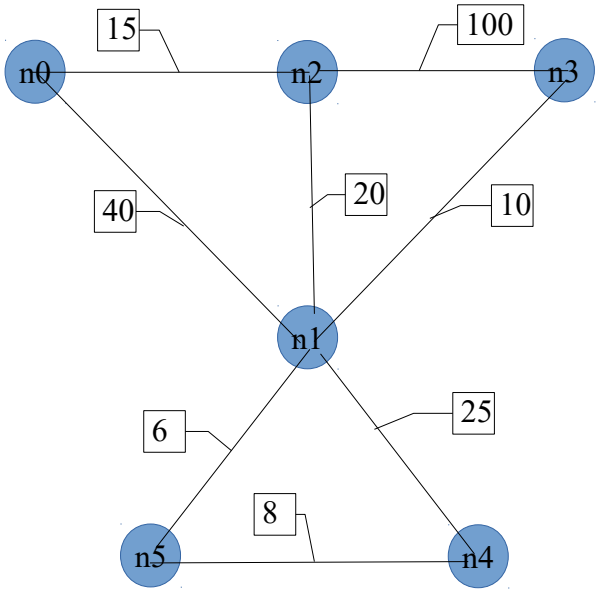
13.3 Instalar cualquier componente que se requiera mediante, en este caso se requiere "numpy":

```
> pip install numpy # o cualquier otro nombre del componente
```

13.4 Finalmente para compilar y ejecutar el programa suponiendo que el nombre del archivo es "prueba_grafo_pesado.py":

```
> python prueba_grafo_pesado.py
```

14. A continuación, usted agregará un método que encuentre el costo más bajo desde un vértice origen hacia todos los demás, así como el camino más corto entre ese vértice origen y todos los demás. Para explicar mejor este método se usará el siguiente grafo pesado:



Matriz de adyacencias (n == 6):

	0	1	2	3	4	5
0	0	40	15	float('inf')	float('inf')	float('inf')
1	40	0	20	10	25	6
2	15	20	0	100	float('inf')	float('inf')
3	float('inf')	10	100	0	float('inf')	float('inf')
4	float('inf')	25	float('inf')	float('inf')	0	8
5	float('inf')	6	float('inf')	float('inf')	8	0

Por ejemplo, un camino fácil de encontrar entre el vértice cero y el vértice 1 es directo, con un peso o costo de 40. Sin embargo, también se puede llegar del vértice cero al uno pasando por el 2, y el costo agregado es apenas 35, siendo menor sería preferible al anterior. De manera similar sucede con el camino más corto entre el vértice cero y el vértice 4: podría ser 0 --> 1 --> 4 con costo de 65, pero resulta mejor 0 --> 2 --> 1 --> 4 con costo de 60. Se entiende entonces que el segundo es un camino "más corto" que el primero o "menos costoso".

El método a elaborar debe encontrar el costo más bajo y el camino más corto entre un vértice dado como origen y todos los demás del grafo. Para esto se deberá usar el algoritmo clásico de Dijkstra, tal como ha sido descrito en: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

El nuevo método tendrá la siguiente especificación, encabezado o firma:

```
    # REQ: 0 <= idvrt < n.
    # EFE: retorna un tuple con dos arreglos:
    #             1. dist[] contiene n - 1 valores, cada uno con el costo del
    #             camino más corto desde el origen hacia los demás vértices.
    #             2. prev[] contiene n - 1 valores identificadores de vértices.
    #             prev[i] contiene el identificador del vértice antecesor en
    #             el camino más corto desde el origen hacia el vértice cuyo identificador
    #             es i.
    def obt_caminos_mas_cortos(origen):
           # su codigo
           return dist, prev    # dos arreglos o listas
```

El algoritmo a implementar aparece en el url indicado de la Wikipedia:

```
 1  function Dijkstra(Graph, source):
 2
 3      create vertex set Q
 4
 5      for each vertex v in Graph:
 6          dist[v] ← INFINITY
 7          prev[v] ← UNDEFINED
 8          add v to Q
10      dist[source] ← 0
11
12      while Q is not empty:
13          u ← vertex in Q with min dist[u]
14
15          remove u from Q
16
17          for each neighbor v of u:           // only v that are still in Q
18              alt ← dist[u] + length(u, v)
19              if alt < dist[v]:
20                  dist[v] ← alt
21                  prev[v] ← u
22
23      return dist[], prev[]
```

15. Antes de continuar, deberá agregar un inicializador para que pueda realizar pruebas con el ejemplo de la página de anterior. Escriba un inicializador que llene casilla por casilla la matriz 6 x 6 para el grafo del ejemplo. Este método inicializador deberá ser invocado desde la función "m ain()" o controladora después de haber creado la nueva instancia:

```
import random
from grafo_pesado import GrafoPesado

def main():
       # construye un grafo con n == 6
       mi_grafo = GrafoPesado(6)

       # inicializa el grafo previamente construido con n == 6
       # con las aristas y los pesos del ejemplo.
       mi_grafo.init__ejemplo()
       return

main()
```