Gestión de la Información en la Web
Curso 2024–25
Práctica 1 — Python

Fecha de entrega: domingo 6 de octubre

Entrega de la práctica
La entrega de la práctica se realizará a través del Campus Virtual de la asignatura mediante un fichero `pr1.py`. El esqueleto de este fichero se puede descargar del Campus Virtual.

Lenguaje de programación
Python 3.11 o superior.

Calificación
Se medirá la corrección mediante tests de unidad. Además de la corrección, se valorará la **calidad, concisión y claridad del código**, la incorporación de **comentarios** explicativos, su **eficiencia** tanto en tiempo como en memoria y la puntuación obtenida en Pylint.

Declaración de autoría e integridad
Todos los ficheros entregados contendrán una cabecera en la que se indique la asignatura, la práctica, el grupo y los autores. Esta cabecera también contendrá la siguiente declaración de integridad:

> *Declaramos que esta solución es fruto exclusivamente de nuestro trabajo personal. No hemos sido ayudados por ninguna otra persona o sistema automático ni hemos obtenido la solución de fuentes externas, y tampoco hemos compartido nuestra solución con otras personas de manera directa o indirecta. Declaramos además que no hemos realizado de manera deshonesta ninguna otra actividad que pueda mejorar nuestros resultados ni perjudicar los resultados de los demás.*

No se corregirá ningún fichero que no venga acompañado de dicha cabecera.

En esta práctica implementaremos algunas funciones en Python para dominar las distintas construcciones del lenguaje y sus tipos de datos más importantes (listas, diccionarios, tuplas, conjuntos).

1. Manejo básico de matrices [4pt]

Implementar una serie de funciones para el manejo de matrices en Python. En este ejercicio consideraremos que las matrices se representan como listas de listas. Por ejemplo, las matrices

$$A = \begin{pmatrix} 1 & 0 & 2 & 5 \\ 0 & 3 & 3 & 5 \\ 1 & 2 & 2 & 5 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 3 & 2 \\ 1 & 2 & 2 \end{pmatrix}$$

se representarían como **listas de listas**, donde cada lista interna es una fila:

```
a = [[1, 0, 2, 5],          b = [[1, 0, 1],
     [0, 3, 3, 5],               [0, 3, 2],
     [1, 2, 2, 5]]               [1, 2, 2]]
```

Las operaciones que hay que implementar son:

- `dimension(matriz)` devuelve una tupla `(filas, columnas)` con el tamaño de la matriz. Si la matriz está mal formada (no tiene ninguna fila, o las filas son de distinto tamaño) deberá devolver `None`.

  Ejemplo:

```
>>> dimension(a)
(3, 4)
>>> dimension(b)
(3, 3)
```

- `es_cuadrada(matriz)` devuelve `True/False` indicando si la matriz es cuadrada.

  Ejemplo:

```
>>> es_cuadrada(a)
False
>>> es_cuadrada(b)
True
```

- `es_simetrica(matriz)` devuelve `True/False` indicando si la matriz es simétrica (consultar `https://es.wikipedia.org/wiki/Matriz_sim%C3%A9trica` si necesitáis recordar esta definición).

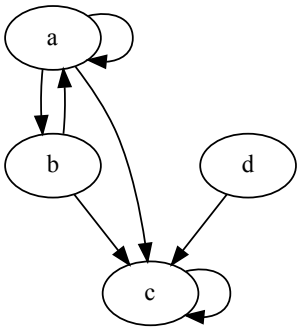
  Ejemplo:

```
>>> es_simetrica(a)
False
>>> es_simetrica(b)
True
```

- `multiplica_escalar(matriz, k)` devuelve **una nueva matriz** (sin modificar la matriz pasada como parámetro) con el resultado de multiplicar todos los elementos de la matriz original por el valor `k`. Si la matriz está mal formada debe devolver `None`.

  Ejemplo:

```
>>> multiplica_escalar(a, 2)
[[2, 0, 4, 10],
 [0, 6, 6, 10],
 [2, 4, 4, 10]]
```




```
g = {"nodos": ["a", "b", "c", "d"],
     "aristas": {"a": ["a", "b", "c"],
                 "b": ["a", "c"],
                 "c": ["c"],
                 "d": ["c"]
                }
  }
```

Figura 1: Ejemplo de representación de grafo dirigido

```
>>> multiplica_escalar(b, 2)
[[2, 0, 2],
 [0, 6, 4],
 [2, 4, 4]]
```

- `suma(matriz1, matriz2)` devuelve **una nueva matriz una nueva matriz** (sin modificar la matriz pasada como parámetro) con el resultado de sumar las dos matrices. Si alguna matriz está mal formada o no son de la misma dimensión, debe devolver `None`.

  Ejemplo:

```
>>> suma(a, b)
None
>>> print(pr1.suma(a, a))
[[2, 0, 4, 10],
 [0, 6, 6, 10],
 [2, 4, 4, 10]]
```

Importante: **este es un ejercicio puramente de programación para practicar Python, así que no se puede utilizar ninguna biblioteca de manejo de matrices**. En un escenario real, la mejor opción para operar con matrices es la biblioteca `numpy`.

2. Grafos dirigidos [6pt]

En esta sección vamos a trabajar con grafos dirigidos no valorados representados en Python usando listas y diccionarios. La manera de expresar un grafo será con un diccionario:

```
{"nodos": lista de nodos (sin repetición),
 "aristas": {nodo_origen_1: lista de nodos destino 1,
             nodo_origen_2: lista de nodos destino 2,
             ...
             nodo_origen_N: lista de nodos destino N
             }
```

La figura 1 muestra un grafo dirigido de 4 nodos («a», «b», «c» y «d») y 7 aristas, junto con su representación en Python almacenada en la variable `g`. Sobre estos grafos hay que implementar varias funciones:

2.1. Validación de grafos [1.5 puntos]

La función `validar(grafo)` devuelve `True` si `grafo` está bien construido, y `False` en otro caso. Diremos que un grafo está bien construido si:

- El diccionario contiene exactamente las claves: `'nodos'` y `'aristas'`, ambos en minúsculas.
- `nodos` contiene una lista no vacía de etiquetas de nodos.
- `nodos` no tiene nodos repetidos.
- Los nodos origen que aparecen en `aristas` son nodos definidos en `nodos`.
- Los nodos destino que aparecen en `aristas` son nodos definidos en `nodos`.

Ejemplos de invocaciones a la función `validar`

```
>>> validar(g)
True
>>> validar({"nodos": [1,2], "aristas": {1: [2], 2: [2]}})
True
>>> validar({"nodos": [1,2], "aristas": {1: [2]}})
False
>>> validar({"nodos": [], "aristas": {}})
False
```

2.2. Grado de entrada de un nodo [1.5pt]

Se necesita una función `grado_entrada(grafo, nodo)` para conocer el grado de entrada de un nodo del grafo. El grado de entrada de un nodo es el número de aristas que llegan al nodo. Si el grafo no es válido o el nodo indicado no existe en el grafo, se deberá devolver -1.

Ejemplos de invocaciones a la función `grado_entrada`

```
>>> grado_entrada(g, "a")
2
>>> grado_entrada(g, "d")
0
>>> grado_entrada(g, "Z")
-1
>>> grado_entrada({"nodos": [1,2], "aristas": {1: [2]}}, "2")
-1
```

2.3. Diccionario de distancias [3pt]

Por último, se va a programar una función `distancia(grafo, nodo)` para devolver un diccionario con las distancias desde el `nodo` al resto de nodos del grafo. Como el grafo es no valorado y las aristas no tienen peso, como noción de distancia entre el nodo «a» y el nodo «b» consideraremos el mínimo número de aristas que hay que recorrer desde «a» para llegar a «b». Por definición, consideraremos que la distancia de un nodo «a» al mismo nodo «a» es 0, y si un nodo «b» no es alcanzable desde «a» entonces la distancia entre «a» y «b» debe ser -1. De manera similar al apartado anterior, si el grafo no es válido o el nodo indicado no existe en el grafo, se deberá devolver `None`.

Ejemplos de invocaciones a `distancia`

```
>>> distancia(g, "a")
{'a': 0, 'b': 1, 'c': 1, 'd': -1}
>>> distancia(g, "b")
{'a': 1, 'b': 0, 'c': 1, 'd': -1}
```

```
>>> distancia(g, "d")
{'a': -1, 'b': -1, 'c': 1, 'd': 0}
>>> distancia(g, "Z")
None
```

3. Pylint

Para la evaluación de la práctica se tendrá en cuenta la puntuación que obtenga vuestro código en la herramienta de análisis estático `pylint`. Ejecutar `pylint` sobre vuestro fichero de texto `pr1.py` es muy sencillo, únicamente debéis escribir lo siguiente en la línea de comandos:

```
$ pylint pr1.py

--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
```

En una práctica tan sencilla como esta, que no hace uso de bibliotecas externas, debería ser muy sencillo obtener una puntuación de 10. Los mensajes de `pylint` son bastante informativos y en caso de no obtener un 10 os resultará sencillo corregir vuestro código, pero no dudéis en preguntarme si tenéis algún problema.