Introducción a la Programación

Guía Práctica 8
Archivos, Pilas, Colas y Diccionarios

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

1. Archivos

Para usar archivos contamos con las funciones: open, close, read, readline, readlines, write. Para más información referirse a la documentación: `https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files`

Ejercicio 1. Implementar en Python:

1. Una función `contar_lineas(in nombre_archivo : str) → int` que cuenta y devuelva la cantidad de líneas de texto del archivo dado.
2. Una función `existe_palabra(in palabra : str, in nombre_archivo : str) → bool` que dice si una palabra existe en un archivo de texto o no
3. Una función `cantidad_apariciones(in nombre_archivo : str, in palabra : str) → int` que devuelve la cantidad de apariciones de una palabra en un archivo de texto

Ejercicio 2. Dado un archivo de texto con comentarios, implementar una función `clonar_sin_comentarios(in nombre_archivo : str)` que toma un archivo de entrada y genera un nuevo archivo que tiene el contenido original sin las líneas comentadas. Para este ejercicio vamos a considerar comentarios como aquellas líneas que tienen un carácter '#'como primer carácter de la línea, o si no es el primer carácter, se cumple que todos los anteriores son espacios.
Ejemplo:

```
# esto es un comentario
    # esto tambien
esto no es un comentario # esto tampoco
```

Ejercicio 3. Dado un archivo de texto, implementar una función `invertir_lineas(in nombre_archivo : str)`, que escribe un archivo nuevo llamado `reverso.txt` que tiene las mismas líneas que el original, pero en el orden inverso.
Ejemplo: si el archivo original es

```
Esta es la primera linea.
Y esta es la segunda.
```

debe generar:

```
Y esta es la segunda.
Esta es la primera linea.
```

Ejercicio 4. Dado un archivo de texto y una frase, implementar una función `agregar_frase_al_final(in nombre_archivo : str, in frase : str)`, que la agregue al final del archivo original (sin hacer una copia).

Ejercicio 5. Dado un archivo de texto y una frase, implementar una función `agregar_frase_al_principio(in nombre_archivo : str, in frase : str)`, que agregue la frase al comienzo del archivo original (similar al ejercicio anterior, sin hacer una copia del archivo).

Ejercicio 6. Implementar una función `listar_palabras_de_archivo(in nombre_archivo : str) → list`, que lea un archivo en modo binario y devuelva la lista de palabras legibles, donde vamos a definir una palabra legible como:

- secuencias de texto formadas por números, letras mayúsculas/minúsculas y los caracteres ' '(espacio) y '_'(guion bajo)
- que tienen longitud >= 5

Una vez implementada la función, probarla con diferentes archivos binarios (.exe, .zip, .wav, .mp3, etc).
Referencia: https://docs.python.org/es/3/library/functions.html#open

> Para resolver este ejercicio se puede abrir un archivo en modo binario 'b'. Al hacer read() vamos a obtener una secuencia de bytes, que al hacer chr(byte) nos va a devolver un carácter correspondiente al byte leído.

Ejercicio 7. Implementar una función `calcular_promedio_por_estudiante(in nombre_archivo_notas : str, in nombre_archivo_promedios : str)`, que lea un archivo de texto separado por comas (comma-separated values, o .csv) que contiene las notas de toda la carrera de un grupo de estudiantes y calcule el promedio final de uno. El resultado se guardará en un archivo "nombre_archivo_promedios.csv"

El archivo tiene el siguiente formato:

```
nro de LU (str), materia (str), fecha (str), nota (float)
```

Para modularizar el código, implementar la función `promedio_estudiante(in nombre_archivo, in lu : str) → float`.

2. Pilas

Ejercicio 8. Implementar una función `generar_nros_al_azar(in cantidad : int, in desde : int, in hasta : int) → Pila[int]` que genere una pila de *cantidad* de números enteros al azar en el rango [*desde*, *hasta*].

Pueden usar la función `random.randint(< desde >, < hasta >)` y la clase `LifoQueue()` que es un ejemplo de una implementación básica:

```
from queue import LifoQueue as Pila

p = Pila()
p.put(1) # apilar
elemento = p.get() # desapilar
p.empty() # vacia?
```

Ejercicio 9. Implementar una función `cantidad_elementos(in p : Pila) → int` que, dada una pila, cuente y devuelva la cantidad de elementos que contiene. No se puede utilizar la función `LifoQueue.qsize()`. Si se usa `get()` para recorrer la pila, esto modifica el parámetro de entrada. Y como la especificación dice que es de tipo `in` hay que restaurarla.

Ejercicio 10. Dada una pila de enteros, implementar una función `buscar_el_maximo(in p : Pila[int]) → int` que devuelva el máximo elemento.

Ejercicio 11. Implementar una función `esta_bien_balanceada(in s : str) → bool` que dado un string con una formula aritmética sobre los enteros, diga si los paréntesis están bien balanceados. Las fórmulas pueden formarse con:

- los números enteros
- las operaciones básicas +, −, x y /
- paréntesis
- espacios

Entonces las siguientes son formulas aritméticas con sus paréntesis bien balanceados:

$1 + (2 \text{ x } 3 - (20 / 5))$
$10 * (1 + (2 * (-1)))$

Y la siguiente es una formula que no tiene los paréntesis bien balanceados:

$1 +) \ 2 \text{ x } 3 \ (\ ()$

Ejercicio 12. La notación polaca inversa, también conocida como notación postfix, es una forma de escribir expresiones matemáticas en la que los operadores siguen a sus operandos. Por ejemplo, la expresión "3 + 4" se escribe como "3 4 +" en notación postfix. Para evaluar una expresión en notación postfix, se puede usar una pila.

Escribe una función `evaluar_expresion(in s : str) → float` en Python que tome una expresión en notación postfix y la evalúe. La expresión estará representada como una cadena de caracteres, donde los operandos y operadores estarán separados por espacios. Para simplificar el problema, sólo vamos a considerar los operandos '+', '-', '*' y '/'(suma, resta, multiplicación y división), los operadores son números.

Para resolver este problema, se recomienda seguir el siguiente algoritmo:

1. Dividir la expresión en tokens (operandos y operadores) utilizando espacios como delimitadores.
2. Recorre los tokens uno por uno.
   a) Si es un operando, agrégalo a una pila.
   b) Si es un operador, saca los dos operandos superiores de la pila, aplícale el operador y luego coloca el resultado en la pila.
3. Al final de la evaluación, la pila contendrá el resultado de la expresión.

Ejemplo de uso:

```
expresion = "3 4 + 5 * 2 -"
resultado = evaluar_expresion(expresion)
print(resultado)  # Debería devolver 33
```

3. Colas

Ejercicio 13. Usando la función `generar_nros_al_azar()` definida en la sección anterior, implementar una función que arme una cola de enteros con los números generados al azar. Pueden usar la clase `Queue()` que es un ejemplo de una implementación básica:

```
from queue import Queue as Cola

c = Cola()
c.put(1) # encolar
elemento = c.get() # desencolar()
c.empty() # vacia?
```

Ejercicio 14. Implementar una función `cantidad_elementos(in c : Cola) → int` que, dada una , cuente y devuelva la cantidad de elementos que contiene. Comparar con la versión usando pila. No se puede utilizar la función `Queue.qsize()`.

Ejercicio 15. Dada una cola de enteros, implementar una función `buscar_el_maximo(in c : Cola[int]) → int` que devuelva el máximo elemento. Comparar con la versión usando pila.

Ejercicio 16. Bingo: un cartón de bingo contiene 12 números al azar en el rango $[0, 99]$.

1. implementar una función `armar_secuencia_de_bingo() → Cola[int]` que genere una cola con los números del 0 al 99 ordenados al azar.

2. implementar una función `jugar_carton_de_bingo(in carton : list[int], in bolillero : Cola[int]) → int` que toma un cartón de Bingo y una cola de enteros (que corresponden a las bolillas numeradas) y determina cual es la cantidad de jugadas de ese bolillero que se necesitan para ganar.

Ejercicio 17. Vamos a modelar una guardia de un hospital usando una cola donde se van almacenando los pedidos de atención para los pacientes que van llegando. A cada paciente se le asigna una prioridad del 1 al 10 (donde la prioridad 1 es la mas urgente y requiere atención inmediata) junto con su nombre y la especialidad médica que le corresponde.

Implementar la función `n_pacientes_urgentes(in c : Cola[(int, str, str)]) → int` que devuelve la cantidad de pacientes de la cola que tienen prioridad en el rango $[1, 3]$.

Ejercicio 18. La gerencia de un banco nos pide modelar la atención de los clientes usando una cola donde se van registrando los pedidos de atención. Cada vez que ingresa una persona a la entidad, debe completar sus datos en una pantalla que está a la entrada: Nombre y Apellido, DNI, tipo de cuenta (si es preferencial o no) y si tiene prioridad por ser adulto +65, embarazada o con movilidad reducida (prioridad si o no).

La atención a los clientes se da por el siguiente orden: primero las personas que tienen prioridad, luego las que tienen cuenta bancaria preferencial y por último el resto. Dentro de cada subgrupo de clientes, se respeta el orden de llegada.

1. Dar una especificación para el problema planteado.
2. Implementar `atencion_a_clientes(in c : Cola[(str, int, bool, bool)]) → Cola[(str, int, bool, bool)]` que dada la cola de ingreso de clientes al banco devuelve la cola en la que van a ser atendidos.

4. Diccionarios

Para esta sección vamos a usar el tipo `dict` que nos provee python:

Ejercicio 19. Leer un archivo de texto y agrupar la cantidad de palabras de acuerdo a su longitud. Implementar la función `agrupar_por_longitud(in nombre_archivo : str) → dict`

que devuelve un diccionario {`longitud_en_letras : cantidad_de_palabras`}.

Ej el diccionario

```
{
        1: 2,
        2: 10,
        5: 4
}
```

indica que se encontraron 2 palabras de longitud 1, 10 palabras de longitud 2 y 4 palabras de longitud 5. Para este ejercicio vamos a considerar palabras a todas aquellas secuencias de caracteres que no tengan espacios en blanco.

Ejercicio 20. Volver a implementar la función que calcula el promedio de las notas de los alumnos (ejercicio 7), pero ahora devolver un diccionario {`libreta_universitaria : promedio`} con los promedios de todos los alumnos. La entrada será el nombre del archivo donde tenemos los datos a leer.

`calcular_promedio_por_estudiante(in nombre_archivo_notas : str → dict[str, float]`

Ejercicio 21. Implementar la función `la_palabra_mas_frecuente(in nombre_archivo : str) → str` que devuelve la palabra que más veces aparece en un archivo de texto. Se aconseja utilizar un diccionario de palabras para resolver el problema.

Ejercicio 22. Nos piden desarrollar un navegador web muy simple que debe llevar un registro de los sitios web visitados por los usuarios del sistema. El navegador debe permitir al usuario navegar hacia atrás y hacia adelante en la historia de navegación.

1. Crea un diccionario llamado `historiales` que almacenará el historial de navegación para cada usuario. Las claves del diccionario serán los nombres de usuario y los valores serán pilas.
2. Implementar `visitar_sitio(inout: historiales:dict[str, Pila[str]], in: usuario:str, in:sitio:str)` que reciba el diccionario de historiales, el nombre de usuario y el sitio web visitado. La función debe agregar el sitio web al historial del usuario correspondiente.

3. Implementar `navegar_atras(inout: historiales: dict[str, Pila[str]], in: usuario:str)` que permita al usuario navegar hacia atrás en la historia de navegación. Primero debemos obtener el sitio anterior al actual, y luego poner este como último en la pila.

Ejemplo de uso:

```
historiales = {}
visitar_sitio(historiales, "Usuario1", "google.com")
visitar_sitio(historiales, "Usuario1", "facebook.com")
navegar_atras(historiales, "Usuario1")
visitar_sitio(historiales, "Usuario2", "youtube.com")
```

Ejercicio 23. Nos piden desarrollar un sistema de gestión de inventario para una tienda de ropa. Este sistema permite llevar un registro de los productos en el inventario y realizar operaciones como agregar nuevos productos, actualizar las existencias y calcular el valor total del inventario.

Para resolver este problema vamos a utilizar un diccionario llamado `inventario` que almacene información sobre los productos. Cada elemento del diccionario debe tener el nombre del producto como clave y otro diccionario con información adicional (nombre, precio y cantidad) como valor.

Agregar en las funciones los tipos de datos correspondientes.

1. Implementa una función llamada `agregar_producto(inout inventario, in nombre, in precio, in cantidad)` que permita agregar un nuevo producto al inventario. El nombre del producto debe ser la clave del diccionario, y el valor debe ser otro diccionario con las claves "precio" y "cantidad". Como requisito de esta función el producto a agregar no está en el inventario.

2. Implementa una función llamada `actualizar_stock(inout inventario, in nombre, in cantidad)` que permita actualizar la cantidad de un producto existente en el inventario.

3. Implementa una función llamada `actualizar_precios(inout inventario, in nombre, in precio)` que permita actualizar el precio de un producto existente en el inventario.

4. Implementa una función llamada `calcular_valor_inventario(in inventario)` $\rightarrow$ `float` que calcule el valor total del inventario multiplicando el precio por la cantidad de cada producto y sumando los valores de todos los productos.

Ejemplo de uso:

```
inventario = {}
agregar_producto(inventario, "Camisa", 20.0, 50)
agregar_producto(inventario, "Pantalón", 30.0, 30)
actualizar_existencias(inventario, "Camisa", 10)
valor_total = calcular_valor_inventario(inventario)
print("Valor total del inventario:", valor_total) # Debería imprimir 1300.00
```