Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 – Lenguajes de Programación I
Septiembre–Diciembre 2025

Estudiante: Luis Miguel Isea Pérez
Carnet: 19-10175

Examen 2

(20 puntos)

2 de noviembre de 2025

Índice

Pregunta 1

(a) Breve descripción del lenguaje escogido

Mi primer apellido es Isea, lo que me deja con I como la inicial para el lenguaje, pero después de investigar bastante no pude encontrar un lenguaje de alto nivel y de propósito general lo suficientemente intuitivo y versátil para poder hacer muchos de los programas pedidos en este examen. Estuve considerando Icon por un tiempo e intentando escribir los programas pedidos, pero no podía lograr los resultados, por lo que decidí usar mi segundo apellido, que es Pérez. Naturalmente, como tengo de inicial la P, decidí ir con Python.

Python es un lenguaje de programación interpretado de alto nivel con tipado dinámico. Fue creado por Guido van Rossumy en 1991, y como curiosidad el nombre Python viene de *Monty Python*, que es básicamente una comedia británica que le gustaba mucho a Rossumy.

Lo que caracteriza a Python es su flexibilidad y facilidad de aprender para los usuarios, con sintaxis bastante sencilla, lo que permite que su tiempo de desarrollo sea mucho más corto, aunque esto hace que sus programas no sean tan rápidos en el tiempo de ejecución. Con esto me refiero al tipado dinámico, el dynamic binding, que sea interpretado en vez de compilado, que en vez de llaves y bloques, se use simplemente la indentación para especificar los niveles de profundidad dentro del código.

Otra de las principales ventajas y fortalezas de Python es que soporta la programación orientada a objetos, la programación imperativa e incluso la programación funcional, lo que lo vuelve un lenguaje multiparadigma.

I. Estructuras de control de flujo

Python ofrece las siguientes estructuras de control de flujo:

Estructuras condicionales

Las estructuras condicionales permiten ejecutar diferentes bloques de código evaluando si ciertas expresiones booleanas son true o false.

- **if-elif-else:** Evalúa condiciones en orden. Si la condición del `if` es verdadera, ejecuta su bloque. Si no, evalúa los `elif` en orden, y si ninguna condición se cumple, ejecuta el bloque `else`.
- **Operador ternario:** Forma compacta, en una sola línea, para un simple if-else.

```python
# if-elif-else: solo se ejecuta un bloque
if edad < 13:
    print("Niño")         # Si edad < 13
elif edad < 18:
    print("Adolescente") # Si 13 <= edad < 18
else:
    print("Adulto")       # Si edad >= 18

# Operador ternario: if-else en forma compacta
resultado = "Aprobado" if nota >= 60 else "Reprobado"
```

Listing 1: Estructuras condicionales en Python

Estructuras iterativas

Permiten repetir la ejecución de un bloque de código múltiples veces.

- **for:** Itera sobre los elementos de una secuencia (lista, rango, string, etc.). Ejecuta el bloque una vez por cada elemento.
- **while:** Repite el bloque mientras la condición sea verdadera. La condición se evalúa antes de cada iteración.

```python
# for: itera sobre una secuencia
for i in range(3):       # Se ejecuta 3 veces: i=0,1,2
    print(i)

# while: repite mientras la condición sea True
contador = 0
while contador < 3:      # Se ejecuta con contador=0,1,2
    print(contador)
    contador += 1
```

Listing 2: Estructuras iterativas en Python

Estructuras de control de excepciones

Manejan errores y situaciones excepcionales durante la ejecución.

- **try-except:** Intenta ejecutar el bloque `try`. Si ocurre una excepción, ejecuta el bloque `except` correspondiente.
- **else:** Se ejecuta solo si no hubo excepciones en el `try`.
- **finally:** Siempre se ejecuta, haya o no habido excepciones.

```python
try:
    numero = int("123")     # Intenta convertir a entero
except ValueError:
    print("No es número")   # Solo si hay ValueError
else:
    print("Conversión exitosa")  # Solo si NO hay error
finally:
    print("Siempre se ejecuta")  # Siempre se ejecuta
```

Listing 3: Manejo de excepciones en Python

Estructuras de control de contexto

Gestionan recursos que necesitan apertura y cierre (archivos, conexiones, etc.).

- **with:** Garantiza que los recursos se liberen correctamente al salir del bloque, incluso si ocurren excepciones.

```python
# with: el archivo se cierra automáticamente
with open("datos.txt", "r") as archivo:
    contenido = archivo.read()
# Aquí el archivo ya está cerrado automáticamente
```

Listing 4: Context managers en Python

II. Orden de evaluación de expresiones y funciones

Evaluación de expresiones

En Python, el orden de evaluación es el siguiente:

- **Expresiones aritméticas:** Se evalúan según precedencia de operadores (paréntesis, potencias, multiplicación/división, suma/resta) y de izquierda a derecha para operadores de igual precedencia.
- **Llamadas a funciones:** Los argumentos se evalúan completamente de izquierda a derecha antes de ejecutar la función.
- **Asignaciones:** El lado derecho se evalúa completamente antes de asignar a la variable.

```python
# Precedencia: multiplicación antes que suma
resultado = 2 + 3 * 4    # 3*4=12, luego 2+12=14

# Izquierda a derecha en misma precedencia
resultado = 10 - 3 - 2   # (10-3)=7, luego 7-2=5

# Argumentos de izquierda a derecha
def suma(a, b, c):
    return a + b + c

# Se evalúa: primero 1+1=2, luego 2*2=4, luego 3+3=6
resultado = suma(1+1, 2*2, 3+3)  # suma(2, 4, 6)
```

Listing 5: Orden de evaluación en Python

Tipo de evaluación

¿Tiene evaluación normal o aplicativa? ¿Tiene evaluación perezosa? Python utiliza principalmente **evaluación aplicativa (eager evaluation)**, lo que significa que las expresiones se evalúan inmediatamente cuando se encuentran en el flujo de ejecución.

Sin embargo, en ciertos contextos específicos, Python emplea **evaluación perezosa (lazy evaluation)**:

```python
# Evaluación aplicativa (predeterminada)
x = 5 + 3               # Se evalúa inmediatamente a 8
y = len([1, 2, 3])      # Se evalúa inmediatamente a 3

# Evaluación perezosa (en contextos específicos)

# 1. Generadores - evalúan bajo demanda
def numeros_pares():
    n = 0
    while n < 6:
        yield n         # Solo produce valor cuando se pide
        n += 2

pares = numeros_pares() # No ejecuta el código interno
print(next(pares))      # Ahora evalúa y devuelve 0
print(next(pares))      # Evalúa y devuelve 2

```

```python
# 2. Operadores de cortocircuito
def funcion_importante():
    print("Ejecutando función...")
    return True

# Cortocircuito: si el primero es False, no evalúa el segundo
resultado = False and funcion_importante()  # No imprime nada

# 3. Expresiones lambda
duplicar = lambda x: x * 2    # No se evalúa al definir
valor = duplicar(5)           # Se evalúa al llamar: 10
```

Listing 6: Tipos de evaluación en Python

Orden de evaluación de argumentos/operandos Python evalúa siempre los argumentos de las funciones de izquierda a derecha:

```python
def mostrar_orden(nombre, valor):
    print(f"{nombre}: {valor}")
    return valor

# Los argumentos se evalúan estrictamente de izquierda a derecha
resultado = mostrar_orden("A", 1) + mostrar_orden("B", 2)

# Salida garantizada:
# A: 1
# B: 2
# resultado = 3

# En llamadas anidadas, mismo orden izquierda-derecha
def combinar(a, b):
    return a + b

resultado = combinar(
    mostrar_orden("Primero", 10),
    mostrar_orden("Segundo", 20)
)
# Salida:
# Primero: 10
# Segundo: 20
```

Listing 7: Orden de evaluación de argumentos

Resumen de evaluación en Python

- **Evaluación principal:** Aplicativa (eager evaluation) - evalúa inmediatamente
- **Evaluación perezosa:** En generadores, operadores `and`/`or` (cortocircuito), y expresiones lambda
- **Orden de argumentos:** Siempre de izquierda a derecha
- **Precedencia:** Sigue reglas matemáticas estándar
- **Cortocircuito:** `and` se detiene en el primer `False`, `or` se detiene en el primer `True`

(b) Implemente los siguientes programas en el lenguaje escogido

Los programas fueron implementados y se subieron al siguiente repositorio de GitHub.

Repositorio GitHub

`https://github.com/lmisea/lenguajes-prog/tree/main/examen2/pregunta1`

Algoritmo Mergesort - Explicación de la implementación

El algoritmo Mergesort implementado sigue la estrategia **"divide y vencerás"** y consta de dos partes principales:

1. División recursiva:

- Se divide la lista en dos mitades aproximadamente iguales
- Se aplica recursivamente Mergesort a cada mitad
- El proceso continúa hasta llegar a listas de 0 o 1 elemento (caso base)

2. Mezcla ordenada:

- Se combinan dos listas ya ordenadas en una nueva lista ordenada
- Se comparan los primeros elementos de cada lista y se toma el menor
- Se repite este proceso hasta agotar ambas listas

Explicación paso a paso:

1. **Caso base:** Si la lista tiene 0 o 1 elemento, ya está ordenada
2. **División:** Se calcula el punto medio y se divide en dos sublistas
3. **Ordenamiento recursivo:** Se aplica Mergesort a cada sublista
4. **Mezcla:** Se combinan las dos sublistas ordenadas en una sola

La implementación utiliza recursión para manejar la división natural del problema y la función `mezclar()` para combinar eficientemente las soluciones parciales.

Pregunta 2

El programa fue implementados y se subió al siguiente repositorio de GitHub.

Repositorio GitHub

`https://github.com/lmisea/lenguajes-prog/tree/main/examen2/pregunta2`

Pregunta 3

Pregunta 3

(a) Análisis del iterador `suspenso`

Dados los valores $X = 1$, $Y = 7$, $Z = 5$, analizamos el siguiente código:

```
for x in suspenso(X + Y + Z, [X, Y, Z]):
    print(x)
```

Donde $X + Y + Z = 1 + 7 + 5 = 13$ y la lista es $[1, 7, 5]$.

Ejecución paso a paso:

1. **Llamada inicial:** `suspenso(13, [1, 7, 5])`
   - $b \neq []$, entonces: `yield 13 + 1 = 14`
   - Llamada recursiva: `suspenso(1, [7, 5])`
2. **Segunda llamada:** `suspenso(1, [7, 5])`
   - $b \neq []$, entonces: `yield 1 + 7 = 8`
   - Llamada recursiva: `suspenso(7, [5])`
3. **Tercera llamada:** `suspenso(7, [5])`
   - $b \neq []$, entonces: `yield 7 + 5 = 12`
   - Llamada recursiva: `suspenso(5, [])`
4. **Cuarta llamada:** `suspenso(5, [])`
   - $b == []$, entonces: `yield 5`

Salida del programa:

```
14
8
12
5
```

(b) Análisis del iterador `misterio`

Analizamos el siguiente código:

```
for x in misterio(5):
    print(x)
```

Ejecución paso a paso:

1. `misterio(5)` llama a `misterio(4)`
2. `misterio(4)` llama a `misterio(3)`

3. `misterio(3)` llama a `misterio(2)`
4. `misterio(2)` llama a `misterio(1)`
5. `misterio(1)` llama a `misterio(0)`
6. **Caso base:** `misterio(0)` yield $[1]$
7. `misterio(1)` procesa $x = [1]$:
   - $r = []$
   - `suspenso(0, [1])` produce: $0 + 1 = 1$ y luego 1
   - $r = [1, 1]$
   - yield $[1, 1]$
8. `misterio(2)` procesa $x = [1, 1]$:
   - $r = []$
   - `suspenso(0, [1, 1])` produce: $0 + 1 = 1$, $1 + 1 = 2$, 1
   - $r = [1, 2, 1]$
   - yield $[1, 2, 1]$
9. `misterio(3)` procesa $x = [1, 2, 1]$:
   - $r = []$
   - `suspenso(0, [1, 2, 1])` produce: $0 + 1 = 1$, $1 + 2 = 3$, $2 + 1 = 3$, 1
   - $r = [1, 3, 3, 1]$
   - yield $[1, 3, 3, 1]$
10. `misterio(4)` procesa $x = [1, 3, 3, 1]$:
    - $r = []$
    - `suspenso(0, [1, 3, 3, 1])` produce: $0+1 = 1$, $1+3 = 4$, $3+3 = 6$, $3+1 = 4$, 1
    - $r = [1, 4, 6, 4, 1]$
    - yield $[1, 4, 6, 4, 1]$
11. `misterio(5)` procesa $x = [1, 4, 6, 4, 1]$:
    - $r = []$
    - `suspenso(0, [1, 4, 6, 4, 1])` produce: $0 + 1 = 1$, $1 + 4 = 5$, $4 + 6 = 10$, $6 + 4 = 10$, $4 + 1 = 5$, 1
    - $r = [1, 5, 10, 10, 5, 1]$
    - yield $[1, 5, 10, 10, 5, 1]$

Salida del programa:

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
```

(c) Iterador para lista ordenada

La impletencación se subió al repositorio de github disponible en:

Repositorio GitHub

`https://github.com/lmisea/lenguajes-prog/tree/main/examen2/pregunta3`

Explicación de la implementación:

- Se utiliza un **min-heap** para mantener los elementos en orden
- El heap se construye en tiempo $O(n)$ usando el algoritmo de Floyd
- Cada extracción del mínimo toma tiempo $O(\log n)$
- La complejidad total para iterar todos los elementos es $O(n \log n)$
- No se ordena la lista previamente, el ordenamiento ocurre durante la iteración
- Se mantiene una copia de la lista original para no modificarla

Salida del ejemplo:

```
Lista original: [1, 3, 3, 2, 1]
Elementos en orden: 1 1 2 3 3
```

Pregunta 4

La pregunta 4 fue implementada y analizada dentro del repositorio subido a GitHub en:

Repositorio GitHub

`https://github.com/lmisea/lenguajes-prog/tree/main/examen2/pregunta4`

Incluso se agregó un pequeño gráfico y resultados en formato csv.