Universidad Simón Bolívar.

Departamento de Computación y Tecnología de la Información.

CI3641 – Lenguajes de Programación I.

Septiembre – Diciembre 2023.

Estudiante: Keyber Yosnar Sequera Avendaño.

Carnet: 16-11120.

Parte 1:

Escoja algún lenguaje de programación de alto nivel, de propósito general y orientado a objetos que no comience con alguna de sus iniciales (ya sea de alguno de sus nombres o alguno de sus apellidos).

Mi nombre completo : Keyber Yosnar Sequera Avendaño.

Lenguaje seleccionado : Python.

1 -) Explique la manera de crear y manipular objetos que tiene el lenguaje, incluyendo: constructores, métodos, campos, etc.

Respuesta:

En Python, la creación y manipulación de objetos se realiza a través de las clases.

Clases: las clases se definen usando la palabra clave class, seguida del nombre de la clase. Dentro de la clase se pueden definir atributos y métodos.

```
class MiClase:
    atributo = "valor"

    def metodo(self):
        return "Hola Mundo"
```

Constructores: en Python, el método __init__() se llama constructor y se invoca automáticamente cuando se crea un objeto. Se utiliza para inicializar los atributos de la clase.

```
class MiClase:
    atributo = ""
    def __init__(self, valor):
        self.atributo = valor
```

Existen dos tipos de constructores:

- Constructor predeterminado: no acepta ningún argumento. Su definición solo tiene un argumento que es una referencia a la instancia que se está construyendo.
- Constructor parametrizado: acepta argumentos proporcionados por el programador. El primer argumento es una referencia a la instancia que se está construyendo, conocida como self, y el resto de los argumentos son proporcionados por el programador.

```python
class MiClase:
    atributo1 = ""
    atributo2 = ""
    atributo3 = ""

    def __init__(self, valor1, valor2, valor3):
        self.atributo1 = valor1
        self.atributo2 = valor2
        self.atributo3 = valor3
```

Campos: los campos o atributos se utilizan para almacenar datos. Se definen dentro del cuerpo de la clase o dentro del constuctor.

```python
class MiClase:
    atributo = "valor"
```

2 -) Describa el funcionamiento del manejo de memoria, ya sea explícito (new/delete) o implícito (recolector de basura).

Python maneja la memoria tanto de manera implícita como explícita:

- Manejo de memoria implícito: Python realiza automáticamente la asignación y desasignación de memoria utilizando un recolector de basura. El recolector de basura es un proceso en el que el intérprete libera la memoria cuando no está en uso para que esté disponible para otros objetos. Esto se conoce como "limpieza" de objetos de memoria que no ya no están en uso. El manejo de la memoria en Python es realizado por el intérprete en sí, importante tener en cuenta que el usuario no tiene el control sobre él, incluso si manipula regularmente punteros de objetos a bloques de memoria.
- Manejo de memoria explícito: en algunos casos, liberar explícitamente la memoria se vuelve necesario. La palabra clave "del" puede ser utilizada para eliminar objetos y liberar la memoria que ocupan.
  o
```python
x = 10  # Crear un objeto
del x  # Eliminar el objeto
```

Python usa un sistema de conteo de referencias. El conteo de referencias funciona contando el número de veces que un objeto es referenciado por otros objetos en el sistema. Cuando se eliminan las referencias de un objeto, el conteo de referencias para ese objeto se decrementa. Cuando el conteo de referencias llega a cero, el objeto es desasignado.

3 -) Diga si el lenguaje usa asosiación estática o dinámica de métodos y si hay forma de alterar la elección por defecto del lenguaje.

Python usa la asosiación dinámica de métodos. Esto significa que el intérprete de Python determina cuál es el método apropiado para invocar en tiempo de ejecución, basándose en el tipo de objeto en la jerarquía de clases del objeto. En otras palabras, el método específico a ser llamado se determina dinámicamente, basándose en el tipo real del objeto. Esto es diferente de la asociación estática, donde las referencias se resuelven en tiempo de compilación.

Python permite la asosiación estática de métodos mediante el uso de métodos estáticos. Un método estático se vincula a la clase que representa en lugar de estar vinculado a cualquier objeto de cualquier clase. No puede acceder o modificar el estado de la clase. Los métodos estáticos en Python se definen usando el decorador \@staticmethod y no toman un primer argumento implícito. Por ejemplo:

```python
class MiClase:
    @staticmethod
    def mi_metodo_estatico():
        print("Este es un método estático!")
```

mi_metodo_estatico es un método estático, lo que significa que está vinculado a la clase MiClase y no a una instancia de MiClase. Por lo tanto, se puede llamar a este método en la clase en sí, como

MiClase.mi_metodo_estatico(), en lugar de en una instancia de la clase. Este método no puede acceder o modificar el estado de la clase.

4 -) Describa la jerarquía de tipos, incluyendo mecanismos de herencia múltiple (de haberlos), polimorfismo paramétrico (de tenerlo) y manejo de varianzas.

Jerarquía de tipos en Python:

- Números: los números en Python se dividen en números integrales y no integrales.
  - Números integrales: enteros simples, enteros largos y booleanos.
  - Números no integrales: números de punto flotante, números complejos, decimales y fracciones.
- Colecciones: las colecciones en Python pueden ser mutables e inmutables.
  - Secuencias: las secuencias en Python pueden ser mutables e inmutables.
  - Secuencia mutable: listas.
  - Secuencia inmutable: tuplas, cadenas, unicodes.
- Conjuntos: al igual que las secuencias, los conjuntos en Python pueden ser mutables e inmutables.
  - Conjuntos mutables: conjuntos.
  - Conjuntos inmutables: conjuntos congelados.
- Llamables (Callables): estos son tipos a los que se puede hacer la operación de llamada a función. Incluyen funciones definidas por el usuario, generadores, clases, métodos de instancias, instancias de clases, funciones integradas y métodos integrados.
- Singletones: el tipo singleton tiene un solo valor. Python tiene los siguientes tipos singleton: None, NotImplemented, Ellipsis (...).

Mecanismos de herencia múltiple:

Python permite la herencia múltiple, lo que significa que una clase puede heredar de más de una clase base. Por ejemplo:

```
class Objeto1:
    pass

class Objeto2:
    pass

class Objeto3(Objeto1, Objeto2):
    pass
```

En el ejemplo, Objeto3 hereda de las clases Objeto1 y Objeto2.

Python proporciona una forma de manejar la herencia múltiple mediante el uso de la Orden de Resolución de métodos (MRO). MRO es el orden en que que Python busca métodos y atributos. Si varias clases tienen un método con el mismo nombre, Python lo buscará en el orden de las clases base. Por ejemplo si tenemos:

```python
class Objeto1:
    def test(self):
        print("Objeto1")

class Objeto2:
    def test(self):
        print("Objeto1")

class Objeto3(Objeto1, Objeto2):
    pass

objeto = Objeto3()
objeto.test()
```

La salida será Objeto1 por Objeto1 aparece antes que Objeto2 en la lista de clases base.

También, Python proporciona la función super() que se puede usar para llamar a métodos en clases base. En el contexto de la herencia múltiple, super() puede usarse para llamar a método sde todas las clases base.

Polimorfismo paramétrico:

El polimorfismo paramétrico no es un concepto que se aplique directamente en Python, ya que Python es un lenguaje de tipado dinámico. Sin embargo, Python ofrece una funcionalidad similar a través de su capacidad para manejar diferentes tipos de datos y a través del uso de funciones genéricas. En Python este se logra a través del tipado dinámico y el concepto de "duck typing", donde el tipo de un objeto es menos importante que los métodos y propiedades que el objeto que soporta. Por ejemplo, se puede escribir una función en Python que pueda tomar cualquier objeto que soporte un método u operación particular, estilo:

```python
def obtener_longitud(item):
    return len(item)
```

Esta función obtener_longitud puede tomar cualquier objeto que soporte la operación len(), como una lista, una cadena, un diccionario, etc. Esto es similar al concepto de polimorfismo paramétrico, ya que permite que la función maneje diferentes tipos de datos, pero aunque Python puede ofrecer una funcionalidad similar al polimorfismo paramétrico, no es exactamente lo mismo debido a las diferencias entre los lenguajes de tipado estático y dinámico.

Parte 2:

Escoja algún lenguaje de programación de alto nivel y de propósito general cuyo nombre tenga la misma cantidad de caracteres que su primer nombre.

Nombre : Keyber (6 caracteres).

Lenguaje seleccionado : Python (6 caracteres).

1 -) Diga si su lenguaje provee capacidades nativas para concurrencia, usa librerías o epende de herramientas externas.

La concurrencia en Python se maneja a través de varios módulos que proporcionan soporte para la ejecución concurrente de código. La elección de qué herramienta usar depende de la tarea a ejecutar (vinculada a CPU o vinculada a E/S) y del estilo preferido de desarrollo (multitarea cooperativa o multitarea apropiativa). Algunas herramientas disponibles son:

- Threading: este módulo permite el paralelismo basado en hilos. Es útil para tareas que están principalmente vincluladas a E/S, donde la capacidad de realizar otras tareas mientras se espera a que se complete una operación de E/S puede mejorar el rendimiento.

- Multiprocessing: este módulo permite el paralelismo basado en procesos. Es útil para tareas que están principalmente vinculadas a la CPU, conde la capacidad de realizar cálculos en múltiples núcleos puede mejorar el rendimiento.
- Asyncio: este módulo permite la multitarea cooperativa. Es útil para tareas que implican muchas operaciones de E/S que se pueden iniciar y luego dejar para que se completen más tarde, mientras que otras tareas se ejecutan en el ínterin.
- Concurrent.futures: este módulo proporciona una interfaz de alto nivel para lanzar tareas paralelas. Puede trabajar con hilos (a través de ThreadPoolExecutor) o con procesos (a través de ProcessPoolExecutor).

2 -) Explique la creación/manejo de tareas concurrentes, así como el control de la memoria compartida y/o pasaje de mensajes.

- Threading:
  - Creación de tareas concurrentes:
    - El módulo threading en Python permite la creación de múltiples hilos de ejecución. Para crear un nuevo hilo, se instancia la clase Thread y se pasa la función objetivo y sus argumentos. Por ejemplo:

```
from threading import Thread

def tarea():
    print("Ejecutando tarea...")

hilo = Thread(target=tarea)
hilo.start()
```

    - La función tarea es la función que se ejecutará en un nuevo hilo. El método start() inicial el hilo.
  - Control de la memoria compartida:
    - En Python, cada hilo teien su propio espacio de memoria. Sin embargo, todos los hilos pueden acceder a la memoria global, lo que permite compartir datos entre hilos. Tener en cuenta que el acceso concurrente a la memoria compartida puede llevar a condiciones de carrera, por lo que a menudo es necesario usar mecanismos de sincronización, como bloqueos, para garantizar que solo un hilo modifique la memoria compartida a la vez.
  - Pasaje de mensajes:
    - El pasaje de mensajes entre hilos en Python se puede lograr de varias maneras, incluyendo colas y tuberías. Una forma común es usar la clase Queue del módulo queue, que proporciona una implementación de cola segura para hilos. Los hilos pueden poner mensajes en la cola y otros hilos pueden obtenerlos, permitiendo la comunicación entre hilos.
- Multiprocessing:
  - Creación de tareas concurrentes:
    - El módulo multiprocessing de Python permite crear y ejecutar procesos que pueden realizar tareas de forma paralela, aprovechando los múltiples núcleos o procesadores de una máquina. El módulo multiprocessing ofrece una API similar a la del módulo threading, pero con algunas diferencias importantes. Por ejemplo:

```
from multiprocessing import Process

def saludo(nombre):
    print("Hola", nombre)

if __name__ == "__main__":
    p = Process(target=saludo, args=("Ana",))
    p.start()
    p.join()
```

    - Este código crea un proceso llamado p que ejecuta la función saludo con el argumento “Ana”. El método start () inicia el proceso y el método join () espera a que termine.
  - Control de la memoria compartida:
    - Para controlar la memoria compartida entre procesos, se puede usar la clase SharedMemory, que permite asignar y administrar bloques de memoria compartida entre uno o más procesos. La clase SharedMemory recibe como argumento el nombre del bloque de memoria compartida y opcionalmente el tamaño en bytes.

- Pasaje de mensajes:
  - Para pasar mensajes entre procesos, se puede usar una Queue para enviar y recibir objetos serializables por medio de mensajes. Una Queue es una cola enlazada que permite insertar objetos al final (put) y extraerlos del principio (get).

- Asyncio:
  - Creación de tareas concurrentes:
    - Para crear tareas concurrentes con Asyncio, se puede usar la clase Task, que representa una tarea asíncrona que se ejecuta en un bucle de eventos. Una tarea puede ser creada a partir de una corutina (una función especializada que usa las palabras clave async y await) o a partir del método run () de otra tarea. Por ejemplo:

```
import asyncio

async def saludo():
    print("Hola")

async def main():
    # Crear una tarea a partir de una corutina
    task1 = asyncio.create_task(saludo())
    # Esperar a que termine la tarea
    await task1
    # Crear otra tarea a partir del método run () de otra tarea
    task2 = saludo()
    # Esperar a que termine la tarea
    await task2

asyncio.run(main())
```

    - Este código crea dos tareas: task1 y task2, que se ejecutan en paralelo usando el bucle de eventos. La primera tarea llama a la función saludo(), que imprime “Hola” y devuelve “saludado”. La segunda tarea llama al mismo método saludo(), pero no devuelve nada. El resultado es “saludado”.
  - Control de la memoria compartida:
    - Para controlar la memoria compartida entre procesos con Asyncio, se puede usar la clase SharedMemory, que permite asignar y administrar bloques de memoria compartida entre uno o más procesos. La clase SharedMemory recibe como argumento el nombre del bloque de memoria compartida y opcionalmente el tamaño en bytes.
  - Pasaje de mensajes:
    - Para pasar mensajes entre procesos con Asyncio, se puede usar una cola para enviar y recibir objetos serializables por medio de mensajes. Una cola es una cola enlazada que permite insertar objetos al final (put) y extraerlos del principio (get).

- Concurrent.futures:
  - Creación de tareas concurrentes:
    - Para crear tareas concurrentes con Concurrent.futures, se puede usar la clase ThreadPoolExecutor, que crea una cola de trabajo donde se pueden enviar las tareas a ejecutar por medio del método submit(). El método submit() recibe como argumento el callable a ejecutar y devuelve un Future que representa la tarea. Por ejemplo:

```
from concurrent.futures import ThreadPoolExecutor

def sumar(a, b):
    return a + b

with ThreadPoolExecutor(max_workers=4) as executor:
    future = executor.submit(sumar, 3, 5)
    print(future.result()) # imprime 8
```

    - Este código crea una cola de trabajo con cuatro hilos y envía la función sumar a ejecutar con los argumentos 3 y 5. El resultado es el valor devuelto por la función sumar.
  - Memoria compartida:
    - Para controlar la memoria compartida entre procesos con Concurrent.futures, se puede usar la clase SharedMemory, que permite asignar y administrar bloques de memoria

compartida entre uno o más procesos. La clase SharedMemory recibe como argumento el nombre del bloque de memoria compartida y opcionalmente el tamaño en bytes.

- Pasaje de mensajes:
  - Para pasar mensajes entre procesos con Concurrent.futures, se puede usar una cola para enviar y recibir objetos serializables por medio de mensajes. Una cola es una cola enlazada que permite insertar objetos al final (put) y extraerlos del principio (get).

3 -) Describa el mecanismo de sincronización que usa el lenguaje.

Algunas de las primitivas de sincronización más comunes son:

- Lock: Implementa un cierre de exclusión mutua para tareas asyncio. No es seguro en hilos. Un cierre asyncio puede usarse para garantizar el acceso en exclusiva a un recurso compartido.
- Event: un objeto de eventos que puede usarse para notificar a múltiples tareas asyncio que ha ocurrido algún evento.
- Condition: una clase que implementa un bloqueo condicional, que permite a una tarea esperar hasta que se cupla una condición determinada por la tarea.
- Semaphore: una clase que implementa un semáforo, que limia el número máximo de tareas que pueden ejecutarse simultáneamente en una cola.
- BoundedSemaphore: una versión de Semaphore que lanza una exceción ValueError si aumenta el contador interno por encima del valor inicial.
- Barrier: una clase que implementa una barrera, que permite a varias tareas esperar hasta que todas las demás hayan completado su trabajo.

Parte 5:

Considere las funciones foldr y const, escritas en un lenguaje muy similar a Haskell:

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr _ e [] = e

foldr f e (x:xs) = f x $ foldr f e xs

const :: a -> b -> a

const x _ = x
```

Considere también la función que aplica una función solamente sobre la cola de una lista y agrupa la cabeza con otro valor dado:

```
what :: a -> ([b]  -> [(a, b) -> [b] -> [(a, b)]])

what _ _ [] = []

what x f (y:ys) = (x, y) : f ys
```

a -) Considere la siguiente implementación de una función misteriosa, usando foldr:

```
misteriosa :: ???

gen n = n : gen (n + 1)
```

Muestre la evaluación, paso a paso, de la expresión misteriosa "abc" (gen 1), considerando que:

- El lenguaje tiene orden de evaluación normal.

```
misteriosa "abc" (gen 1)

= foldr what (const[]) "abc" (gen 1)

= what "a" $ foldr what (const[]) "bc" (gen 1)

= what "a" $ foldr what (const[]) "bc" (1:gen 2)
```

= ("a", 1) : $ foldr what (const[]) "bc" (gen 2)

= ("a", 1) : what "b" $ foldr what (const[]) "c" (gen 2)

= ("a", 1) : what "b" $ foldr what (const[]) "c" (2:gen 3)

= ("a", 1) : ("b", 2) : $ foldr what (const []) "c" (gen 3)

= ("a", 1) : ("b", 2) : what "c" $ foldr what (const[]) "" (gen 3)

= ("a", 1) : ("b", 2) : what "c" $ foldr what (const[]) "" (3 : gen 4)

= ("a", 1) : ("b", 2) : ("c", 3) $ foldr what (const[]) "" (gen 4)

= ("a", 1) : ("b", 2) : ("c", 3) : (const[]) (gen 4)

= ("a", 1) : ("b", 2) : ("c", 3) : []

= ("a", 1) : ("b", 2) : ("c", 3) : []

= ("a", 1) : ("b", 2) : [("c", 3)]

= ("a", 1) : [("b", 2), ("c", 3)]

= [("a", 1), ("b", 2), ("c", 3)]

- El lenguaje tiene orden de evaluación aplicativo.

misteriosa "abc" (gen 1)

= misteriosa "abc" (1:gen 2)

= misteriosa "abc" (1:2:gen 3)

= misteriosa "abc" (1:2:3:gen 4)

= misteriosa "abc" (1:2:3:4:gen 5)

= ...

La evaluación es recursiva infinita del gen y se llamará a si misma siempre, por lo que no termina.

b -) Considere el siguiente tipo de datos que representa árboles binarios con información en las ramas:

data Arbol a = Hoja | Rama a (Arbol a) (Arbol a)

Construya una función foldA (junto con su firma) que perita reducir su valor de tipo (Arbol a) a algún tipo b (de forma análoga a foldr). Su implementación debe poder tratar con estructuras potencialmente infinitas.

Su función debe cumplir con la siguiente firma:

foldA :: (a -> b -> b -> b) -> b -> Arbol a -> b

Respuesta:

foldA :: (a -> b -> b -> b) -> b -> Arbol a -> b

foldA _ b Hoja = b

foldA f b (Rama a i d) = f a (foldA f b i) (foldA f b d)