Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2024-2

Publicación: 27 de Septiembre de 2024
Actividad

Actividad 5

Serialización y manejo de *bytes*

Entrega

- **Lugar:** Repositorio personal de GitHub — Carpeta: `Actividades/AC5`
  El código debe estar en la rama (*branch*) por defecto del repositorio: `main`.
- **Fecha máxima de entrega:** 10 de Octubre 23:59

Introducción: Protocolo Kame-DigiSEKAI

Luego de mucha investigación y el arduo apoyo de ChatGPT, se ha logrado digitalizar las almas de seres vivos e introducirnos al mundo digital. Con este gran avance, **Lily416** solicitó poder proteger a todas sus tortugas en la red. No obstante, hay personas diabólicas que quieren hacerles daño, por ejemplo, alterar la edad que tienen o el nombre con el cuál fueron creciendo.

En vistas de estos terribles sucesos, se ha implementado el "Protocolo Kame-DigiSEKAI" para poder encriptar la información de una tortuga. Ahora, te piden a ti crear un programa capaz de encriptar y desencriptar a las tortugas utilizando este protocolo. De este modo, proteger a la tortugas de las personas diabólicas.

Flujo del programa

Esta actividad consta de 2 partes. La primera parte consiste en implementar diferentes funciones que permitan tomar una instancia de `Tortuga` y, mediante uso serialización y manejo de *bytes*, poder encriptarla. Luego, la segunda parte es tomar una secuencia de bytes y desencriptar para recuperar la isntancia de `Tortuga` original. Ambas partes serán corregidas exclusivamente mediante el uso de *tests*.

Finalmente, debes asegurarte de entregar, como mínimo, el archivo que tenga el *tag* de Entregar en la siguiente sección. Los demás archivos no es necesario subir, pero tampoco se penalizará si se suben al repositorio personal.

Archivos

En el directorio de la actividad encontrarás los siguientes archivos:

- Entregar Modificar `main.py`: Contiene las funciones necesarias para encriptar y desencriptar una `Tortuga`.
- No modificar `clases.py`: Contiene la clase `Tortugas` a utilizar en este proceso.

- No modificar `tests_publicos`: Carpeta que contiene diferentes `.py` para ir probando si lo desarrollado hasta el momento cumple con lo esperado. **En la última hoja del enunciado se encuentra un anexo de cómo ejecutar los *tests* por parte o todos**.

Estructura del programa

Esta actividad consta de dos partes, en las cuales se te pedirá que implementes funciones que permitan encriptar y desencriptar las `Tortugas` siguiendo un protocolo específico.

Protocolo Kame-DigiSEKAI

Encriptación

El protocolo de encriptación Kame-DigiSEKAI se basa en 2 elementos: (1) el mensaje a encriptar (que será una instancia de `Tortugas`) y (2) un rango de valores (inicio, fin). Luego, este protocolo presenta 3 pasos: separar el mensaje original en 2 nuevos mensajes, codificar este rango de números en un *bytearray*, y finalmente concatenar los elementos para construir el mensaje encriptado.

Paso 1: Separar el mensaje

El primer paso de este protocolo es extraer, del mensaje a encriptar, los *bytes* indicados en el rango de números entregado. Para esto, se deben asegurar ciertas condiciones:

- Ambos números deben estar dentro del rango del largo del mensaje a encriptar. Esto implica que el inicio no puede ser menor a 0 y el fin no puede ser mayor o igual al largo del mensaje.
- Para que el rango tenga sentido, el valor de fin debe ser igual o mayor al valor de inicio.

Si la secuencia cumple con ambas condiciones, se deben formar 2 *bytearray*: uno con los *bytes* contenidos en el rango indicado por inicio y fin, ambos inclusives, y otro con el mensaje original, pero protegido por una mascara.

- `m_bytes_rango`: este será el primer *bytearray* a generar. Este *bytearray* corresponde al *chunk* contenido en el rango indicado por inicio y fin, ambos extremos incluidos. Por ejemplo:
  - Si inicio y fin son 0, 3 respectivamente, `m_bytes_rango` corresponderá a los *bytes* en la posición 0, 1, 2 y 3.
  - Si inicio y fin son 2, 5 respectivamente, `m_bytes_rango` corresponderá a los *bytes* en la posición 2, 3, 4 y 5

  Finalmente, si el largo de `m_bytes_rango` es impar, este *bytearray* se deberá invertir. Por ejemplo:
  - Si inicio y fin son 0, 4 respectivamente, `m_bytes_rango` corresponderá a los *bytes* en la posición 4, 3, 2, 1 y 0.
- `m_con_mascara`: este será el segundo *bytearray* a generar. Este será una copia exacta del mensaje original, pero los *bytes* contenidos en el rango entregado por inicio y fin, deberán ser reemplazados por 0, 1, 2 y así sucesivamente. Por ejemplo
  - Si el mensaje original era `ABCD`, inicio es 1 y fin es 2, `m_bytes_rango` quedará como `A01D`.
  - Si el mensaje original era `ABCDEFG`, inicio es 2 y fin es 5, `m_bytes_rango` quedará como `AB0123G`.

A modo de ejemplo, llamemos a `mensaje` la secuencia de *bytes* que se desea encriptar, `inicio` al valor de inicio del rango y `fin` al valor del fin del rango. El resultado de este paso sería la creación de `m_con_mascara` y `m_bytes_rango`, las dos nuevas secuencias de *bytes* generadas. No olvidar que si el largo de `m_bytes_rango` es impar, entonces hay que invertir el *bytearray*.

$$\begin{aligned}
\texttt{mensaje} &= b_0\ b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8\ b_9\ b_{10} \\
\texttt{inicio} &= 1 \\
\texttt{fin} &= 5 \\
\texttt{m_con_mascara} &= b_0\ 0\ 1\ 2\ 3\ 4\ b_6\ b_7\ b_8\ b_9\ b_{10} \\
\texttt{m_bytes_rango} &= b_5\ b_4\ b_3\ b_2\ b_1
\end{aligned}$$

Paso 2: Codificar rango de números

El segundo paso consiste en codificar los 2 números del rango: inicio y fin. Para esto, se debe transformar cada número en un mensaje de 3 *bytes* con formato *big endian*. Luego, concatenarlos en un *bytearray*.

Por ejemplo, llamemos `inicio` al valor de inicio del rango y `fin` al valor del fin del rango. El resultado de este paso sería la creación de `rango_codificado`:

$$\begin{aligned}
\texttt{inicio} &= 1 \\
\texttt{fin} &= 9 \\
\texttt{rango_codificado} &= \texttt{b'\textbackslash x00\textbackslash x00\textbackslash x01\textbackslash x00\textbackslash x00\textbackslash x09'}
\end{aligned}$$

Paso 3: Concatenar mensaje final

El último paso consiste en concatenar todos los *bytearray* generados para tener un único mensaje encriptado. La estructura del mensaje encriptado es:

1. El primer fragmento del mensaje corresponde al largo de los *bytes* extraídos producto del rango indicado. Para esto, se debe transformar este número en un *bytearray* compuesto por 3 *bytes* en formato *big endian*. Usando el ejemplo mostrado en el paso 1 con `mensaje`, `inicio` y `fin`, este fragmento corresponde a transformar el número 5 en un mensaje de 3 *bytes*: `b'\x00\x00\x05'`
2. El segundo fragmento corresponde al *bytearray* con los *bytes* extraídos utilizando el rango de números. Usando el ejemplo mostrado en los pasos anteriores, el segundo fragmento corresponde al `m_bytes_rango`.
3. El tercer fragmento corresponde al *bytearray* con el mensaje original con la mascara aplicada en los *bytes* extraídos. Usando el ejemplo mostrado en los pasos anteriores, el tercer fragmento corresponde al `m_con_mascara`.
4. El último fragmento corresponde al *bytearray* con la secuencia de número codificados siguiendo las instrucciones del paso 2. Usando el ejemplo mostrado en los pasos anteriores, el último fragmento corresponde a la `rango_codificado`.

A modo de resumen, el formato del mensaje a encriptar es:

`largo_secuencia` + `m_bytes_rango` + `m_con_mascara` + `rango_codificado`

Ejemplo de encriptación

Para ayudarte a entender de mejor forma el método de encriptación que debes implementar, utilizaremos como ejemplo el siguiente mensaje de 13 *bytes* y los rangos 2, 6:

```
b'\x05\x08\x03\x02\x04\x03\x05\x09\x05\x09\x01\x0A\xFF'
```

```
inicio = 2
  fin = 6
```

1. El primer paso es separar el mensaje utilizando la secuencia. En este caso, como se van a extraer 5 *bytes*, que es una cantidad impar, `m_bytes_rango` quedará invertido. Por lo tanto, `m_con_mascara` y `m_bytes_rango` quedarán del siguiente modo:

   `m_con_mascara` = `b'\x05\x08\x00\x01\x02\x03\x04\x09\x05\x09\x01\x0A\xFF'`
   `m_bytes_rango` = `b'\x05\x03\x04\x02\x03`

2. El segundo paso es transformar los 2 números del rango en un mensaje de *bytes*. En este caso, `rango_codificado` quedará del siguiente modo:

   `rango_codificado` = `b'\x00\x00\x02\x00\x00\x06'`

3. El último paso consiste en concatenar las secuencias de *bytes* generadas y agregar, al inicio del mensaje, el largo de la secuencia de números (5 en este caso). El mensaje encriptado final será:

   `b'\x05\x00\x00' + m_bytes_rango + m_con_mascara + rango_codificado`

   `b'\x00\x00\x05' + b'\x05\x03\x04\x02\x03 +`
   `b'\x05\x08\x00\x01\x02\x03\x04\x09\x05\x09\x01\x0A\xFF' +`
   `b'\x00\x00\x02\x00\x00\x06'`

Desencriptación

En el caso de la desencriptación, se aplicarán los mismos cambios de la encriptación, pero de forma invertida para obtener el mensaje original.

Ejemplo de desencriptación

Finalmente, para ayudarte a entender el proceso de desencriptación, utilizaremos el siguiente mensaje que corresponde al resultado del ejemplo anterior de encriptación.

```
b'\x00\x00\x05'
b'\x05\x03\x04\x02\x03
b'\x05\x08\x00\x01\x02\x03\x04\x09\x05\x09\x01\x0A\xFF'
b'\x00\x00\x02\x00\x00\x06'
```

1. El primer paso es separar el mensaje en los 4 fragmentos. Para esto, necesitamos el el largo de `m_bytes_rango` y los 2 números del rango con el cual se encriptó el mensaje.
   - En el caso del largo, este siempre estará contenido en los 3 primeros *bytes* del mensaje, en este caso: `b'\x00\x00\x05'`. Luego, se transforman esos *bytes* en el `int` correspondiente: 5.
   - Para el caso de los números del rango, estos siempre ocuparán los últimos 6 *bytes* del mensaje, en este caso: `b'\x00\x00\x02\x00\x00\x06'`. Donde los 3 primeros guardan el valor de inicio y los últimos 3 el valor de fin. Luego, se deben transforman en los `int` correspondiente: 2 y 6.
2. Ya conocido el valor del largo de la secuencia y los rangos, se puede extraer `m_con_mascara` y `m_bytes_rango`. En este caso:
   - Dado que el largo es 5, `m_bytes_rango` serán los 5 *bytes* posteriores a los *bytes* asociadas al largo. Es decir, los *bytes* 3, 4, 5, 6 y 7.[1]
   - Finalmente, el resto del mensaje corresponda el mensaje original con la mascara aplicada a los *bytes* indicados por el rango de números, es decir, `m_con_mascara`.

   `m_bytes_rango` = `b'\x05\x03\x04\x02\x03`
   `m_con_mascara` = `b'\x05\x08\x00\x01\x02\x03\x04\x09\x05\x09\x01\x0A\xFF'`

3. El tercer paso es asegurar que `m_bytes_rango` esté en el orden correcto. Dado que el largo fue impar, implica que este *bytearray* fue invertido al momento de encriptar, por lo tanto hay que volver a invertirlo. Dejando `m_bytes_rango` del siguiente modo:

   `m_bytes_rango` = `b'\x03\x02\x04\x03\x05`

4. El último paso consiste en unificar `m_bytes_rango` y `m_con_mascara` asegurando que los *bytes* de `m_bytes_rango` estén correctamente posicionado en `m_con_mascara` en función de los números indicados por `inicio` y `fin`. En este caso, `m_bytes_rango` debe reemplazar los bytes 2, 3, 4, 5 y 6 de `m_con_mascara`. El resultado final sería:

   `b'\x05\x08\x03\x02\x04\x03\x05\x09\x05\x09\x01\x0A\xFF'`

[1] Recordar que un *bytearray* es al final una lista cuyo primer valor está en la posición 0. Así que los *bytes* asociadas al largo estarían en los *bytes* 0, 1 y 2.

Parte 1 - Encriptación

En esta parte, debes completar todas las funciones necesarias para poder encriptar un mensaje correctamente.

- **Modificar** `def serializar_tortuga(tortuga: Tortuga) -> bytearray:`
  Esta función transforma una instancia `Tortuga` en un *bytearray* mediante el uso de `Pickle`. En caso que ocurra algún error del tipo `AttributeError` durante todo este proceso, se deberá atrapar dicha excepción y levantar otra excepción del tipo `ValueError`.

- **Modificar** `def verificar_rango(mensaje: bytearray, inicio: int, fin: int) -> None:`
  Esta función verifica que el mensaje y el rango (inicio, fin) cumplan con las 2 condiciones indicadas en el "Paso 1: Separar el mensaje" del protocolo. En caso que no se cumpla alguna de estas condiciones, se debe levantar una excepción del tipo `AttributeError`. En otro caso, de no levantar ninguna excepción, esta función retorna `None`.

- **Modificar** `def codificar_rango(inicio: int, fin: int) -> bytearray:`
  Esta función se encarga de realizar el "Paso 2: Codificar rango de números" del protocolo, es decir, transformar ambos números en un *bytearray* donde cada número corresponderá a 3 *bytes* con formato *big endian*. Los primeros 3 *bytes* corresponderán a la transformación del número `inicio`, y los siguientes 3 *bytes* serán del número `fin`. Esta función debe retornar el *bytearray* resultante de esta transformación.

- **Modificar** `def codificar_largo(largo: int) -> bytearray:`
  Esta función se encarga de realizar la primera parte del "Paso 3: Concatenar mensaje final" del protocolo, es decir, transformar un número en un *bytearray* compuesto por 3 *bytes* con formato *big endian*. Debe retornar el *bytearray* con el número transformado.

- **Modificar** `def separar_msg(mensaje: bytearray, inicio: int, fin: int) -> List[bytearray]:`
  Esta función se encarga de aplicar el "Paso 1: Separar el mensaje" del protocolo, es decir, segmentar el `mensaje` en 2 *bytearray*.

  - El primer *bytearray* corresponde a `m_bytes_rango`, es decir, los *bytes* cuya posición están dentro del rango indicado por `inicio` y `fin`. En caso que el largo de este *bytearray* sea impar, esta secuencia debe ser invertida.
  - El segundo *bytearray* corresponde a `m_con_mascara`, es decir, al mensaje original con la máscara aplicada a los *bytes* cuya posición están en el rango indicado por `inicio` y `fin`.

  Esta función retorna una lista en donde el primer elemento corresponde a `m_bytes_rango` y el segundo elemento a `m_con_mascara`.

- **No modificar** `def encriptar(mensaje: bytearray, secuencia: List[int]) -> bytearray:`
  Esta función se encarga de encriptar un mensaje utilizando las funciones definidas previamente.

Parte 2 - Desencriptación

En esta parte, debes completar todas las funciones necesarias para poder desencriptar un mensaje correctamente.

- Modificar `def deserializar_tortuga(mensaje_codificado: bytearray) -> Tortuga:`
  Esta función transforma un `bytearray` en la instancia de `Tortuga` que corresponde mediante el uso de `pickle`. En caso que durante este proceso ocurra algún error del tipo `ValueError`, se deberá atrapar dicha excepción y levantar otra excepción del tipo `AttributeError`. En otro caso, esta función debe retornar la instancia de la `Tortuga` obtenida de la deserialización.
- Modificar `def decodificar_largo(mensaje: bytearray) -> int:`
  Esta función se encarga de obtener los primeros 3 *bytes* del mensaje y lo convierte en el número correspondiente aplicando una transformación del tipo *big endian*. Finalmente, retorna el número obtenido de esta transformación.
- Modificar `def separar_msg_encriptado(mensaje: bytearray) -> List[bytearray]:`
  Esta función utiliza a `decodificar_largo` para obtener el largo de la secuencia de números. Luego, con dicho valor se encarga de separar el mensaje en las 3 fragmentos restantes: `m_bytes_rango`, `m_con_mascara` y `rango_codificado`. Además, se encarga de invertir `m_bytes_rango` si es que originalmente fue invertido. Finalmente retorna una lista donde el primer elemento corresponde a `m_bytes_rango`, el segundo elemento corresponde a `m_con_mascara` y el último elemento corresponde a `rango_codificado`.
- Modificar `def decodificar_rango(rango_codificado: bytearray) -> List[int]:`
  Se encarga de obtener el número de inicio y fin contenidos en `rango_codificado`. Para esto, se deben tomar de a 3 *bytes* y convertirlo en el número correspondiente aplicando una transformación del tipo *big endian*. Debe retornar la lista de los números transformados. El primer elemento de la lista corresponde al número de `inicio` y el segundo número corresponda a `fin`.
- Modificar `def desencriptar(mensaje: bytearray) -> bytearray:`
  Esta función se encarga de utilizar las funciones definidas anteriormente (`decodificar_largo`, `separar_msg_encriptado` y `decodificar_rango`) para obtener el mensaje original a partir del mensaje encriptado. Debe retornar un *bytearray* con el mensaje original.

Notas

- **No puedes hacer *import* de otras librerías externas a las entregadas en el archivo.**
- Se recomienda completar la actividad en el orden del enunciado.
- Recuerda que la ubicación de tu entrega es en **tu repositorio de Git**. En la rama (*branch*) por defecto del repositorio: `main`.
- Recuerda que esta evaluación presenta corrección **automatizada**. Si entregas un código que se cae al momento de correr los *tests*, será evaluado con 0 puntos.
- Se recomienda probar tu código con los *tests* y ejecutando `main.py`, este último se ofrece un pequeño código donde se encripta y desencripta una Tortuga.
- Si aparece un error inesperado, ¡léelo! Intenta interpretarlo y/o buscarlo en *Google*.

Ejecución de *tests*

En esta actividad se provee de varios archivos `.py` los cuáles contiene diferentes *tests* que ayudan a validar el desarrollo de la actividad. Para ejecutar estos *tests*, **primero debes posicionar tu terminal/consola en la carpeta de la actividad (`Actividades/AC5`)**. Luego, desde esta misma, debes escribir el siguiente comando para ejecutar todos los *tests* de la actividad:

- `python3 -m unittest discover tests_publicos -v -b`

En cambio, si deseas ejecutar un subconjunto de *tests*, puedes hacerlo si escribes lo siguiente en la terminal/consola:

- `python3 -m unittest -v -b tests_publicos.test_encriptar`
  Para ejecutar solo el subconjunto de *tests* relacionado al proceso de encriptación.
- `python3 -m unittest -v -b tests_publicos.test_desencriptar`
  Para ejecutar solo el subconjunto de *tests* relacionado al proceso de desencriptación.
- `python3 -m unittest -v -b tests_publicos.test_integracion`
  Para ejecutar solo el subconjunto de *tests* relacionado al proceso completo de encriptar y desencriptar `Tortugas`.

Importante: recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `python`, `py3` o `python3.11`.