Práctica 2.0: Introducción a las prácticas de laboratorio de SAR

Descripción:

En la asignatura SAR vamos a usar el lenguage de programación Python. Aunque es un lenguage fácil de usar, existen unos cuantos detalles que lo hacen diferente. Además, en algunas prácticas de esta asignatura usaremos unos cuantos conceptos que no son básicos. El objetivo de esta práctica es tener el primer contacto con todos estos elementos y facilitar el trabajo a realizar en las siguientes sesiones prácticas.

El/la estudiante tendrá que llevar a cabo los pasos indicados en el siguiente guión.

Material

Para hacer esta práctica se dispone de los siguientes recursos:

1. Fichero con el código fuente del programa para hacer los primeros ejercicios del guión (*intro_lab.py*).

Guión

Ejecutando código fuente de Python

1. Analiza el código del fichero *intro_lab.py*.
```
#!/usr/bin/env python3

import sys

if len(sys.argv) > 1:
    print(sys.argv)
```
2. Ejecuta el código usando el comando `python3 intro_lab.py`.
3. Ejecuta de nuevo el código pasándole unos parámetros: `python3 intro_lab.py estos son cuatro parametros`
4. Ejecuta el código usando la siguiente sintáxis: `./intro_lab.py`
   - Si te da un error será porque el fichero no tiene permisos de ejecución. Dale permiso de ejecución con el siguiente comando, `chmod u+x intro_lab.py`, e inténtalo de nuevo.

Indentación

5. Introduce dentro del cuerpo del *if* el siguiente código y vuelve a ejecutar el programa:
```
for arg in sys.argv:
    print(arg, len(arg))
```
   - Podría resultar que tuvieras problemas con la indentación. La sentencia *if* del código usa tabuladores para indentar y puede que tú ahora hayas introducido espacios en blanco.
6. Seguramente el editor de texto que estás usando para editar sea capaz de mostrar los

espacios en blanco y los tabuladores. Intenta activar esa opción para que sean visibles. Así podrás ver dónde estás usando espacios y dónde tabuladores.

7. Los editores de texto también suelen tener una opción para configurar el comportamiento de la tecla *tab*. Intenta localizar también esa opción en tu editor.

El intérprete de Python

8. Para poner en marcha el intérprete de Python ejecuta el siguiente comando en la línea de comandos: `python3`
9. Escribe cualquier sentencia o expresión en Python, pulsa RETURN y será interpretada. Prueba como ejemplo las siguientes expresiones y otras parecidas que se te ocurran:
   - `7 + 3`
   - `7 == 5`
   - `7 in [3, 25, 7, 11]`
10. Puedes también usar variables (el uso de estructuras de control es más difícil porque seguir las reglas de indentación en el intérprete no es tan sencillo):
    - `x = 7`
    - `print(x)`
    - `x`
    - `x > 10`
    - `x = 'Saludos!'`
    - `len(x)`
    - `x.lower()`
11. Para salir del intérprete hay que llamar a la función *exit()* ó pulsar Ctrl+D.

Cadenas de bytes (El tipo de datos *bytes*)

12. Vuelve a entrar en el intérprete ejecutando de nuevo `python3`.
13. Las cadenas de bytes se crean como las cadenas de caracteres pero añadiendo el carácter 'b' antes de las comillas: `b'una cadena de bytes'`
14. Muchas veces es práctico crear una cadena de bytes en hexadecimal. Escribe la siguiente expresión para definir una cadena de 4 bytes en hexadecimal: `b'\x61\x62\x63\x64'`
    - Como habrás visto, la interpretación de la expresión es distinta. Los caracteres US-ASCII imprimibles que aparezcan en una cadena de bytes Python los mostrará como caracteres. Muchas veces esto nos será de ayuda, pero cuando queremos trabajar con valores hexadecimales no será tan cómodo.
    - Ejecuta `b'\x61\x62\xf0\x64'`. Cuando se mezclan caracteres US-ASCII con caracteres que no lo son o que no son imprimibles, el resultado no es tan fácil de interpretar para nosotros.
15. Siempre puedes usar el método *hex()* para ver los valores en hexadecimal. Ejecuta las siguientes sentencias:
    - `x = b'\x61\x62\xf0\x64'`
    - `x`
    - `x.hex()`
    - `x.hex(' ')`
    - `x.hex(':')`
16. También puede que te resulte más cómodo crear la cadena de bytes con el método *fromhex()* (si añades espacios en blanco entre cada byte no serán tenidos en cuenta):
    - `x = bytes.fromhex('61 62 f0 64')`

- `x = bytes.fromhex('6162f064')`

Codificación de caracteres

17. Las cadenas de caracteres (*strings*) para codificarlas como cadenas de bytes (para que puedan ser almacenadas en un fichero o transferidas a través de la red, por ejemplo) se ha de indicar la codificación (el juego de caracteres) a utilizar. Para ello se ha de utilizar el método *encode()* de Python y *decode()* para lo contrario (de *bytes* a *string*). Ejecuta lo siguiente en el intérprete:
    - `x = 'Python'.encode('ascii')`
    - `x`
    - `x.hex(' ')`
    - `x.decode('ascii')`
18. Prueba ahora alguna cadena que contenga algún carácter que no sea US-ASCII (recuerda que el método *encode()* usa por defecto la codificación UTF-8):
    - `x = 'Iñaki'.encode('ascii')`
    - `x = 'Iñaki'.encode('latin1')`
    - `x`
    - `x.hex(' ')`
    - `len(x)`
    - `x.decode('latin1')`
    - `x[2:].decode('ascii')`
    - `x = 'Iñaki'.encode()`
    - `x`
    - `x.hex(' ')`
    - `len(x)`
    - `x.decode()`
    - `x[1:].decode()`
19. Ahora ejecuta la siguiente expresión: `x[2:].decode()`. Intenta aclarar qué es lo que pasa.

Conversiones entre cadenas de bytes (*bytes*) y números enteros (*int*)

20. Ejecuta las siguientes sentencias:
    - `x = b'\x61\x00'`
    - `x`
    - `i = int.from_bytes(x, byteorder='big')`
    - `i`
21. Fíjate cómo el número `i` proviene de $6x16^3 + 1x16^2 + 0x16^1 + 0x16^0$. Con la ordenación 'little', sin embargo, la interpretación del valor como número entero cambia: $0x16^3 + 0x16^2 + 6x16^1 + 1x16^0$
22. Ejecuta las siguientes sentencias para ver el proceso contrario:
    - `i.to_bytes(2, 'big').hex(' ')`
    - `i.to_bytes(4, 'big').hex(' ')`
    - `i.to_bytes(2, 'little').hex(' ')`
23. Si queremos usar el método *to_bytes()* con un número directamente, tenemos que meterlo entre paréntesis:
    - `5.to_bytes(2, 'big')`
    - `(5).to_bytes(2, 'big')`
    - `(5).to_bytes(2, 'big') == bytes.fromhex('00 05')`
24. A pesar de que en Python existe el tipo cadena de bytes (*bytes*), no existe el tipo byte. Si

tomamos un elemento de una cadena de bytes, el tipo de este es *int*:

- `x = bytes.fromhex('00 05 15')`
- `x[1:]` ó `x[1:].hex(' ')`
- `x[1]`
- `x[1:2]`

25. Como habrás visto las 2 últimas opciones no son lo mismo. `x[1]` es un número entero y sin embargo `x[1:2]` es una cadena de bytes de un único byte.

Operaciones con bits

26. Durante la asignatura también tendremos que trabajar con cadenas o ristras de bits. Python tiene distintas opciones para ello:
    - `bin(5)`
    - `format(5, '#b')`
    - `format(5, 'b')`
27. En estos casos se crea una cadena de caracteres con la cadena de bits representada, con tantos caracteres como bits hagan falta (en función de la posición necesaria del bit a 1 de mayor peso). Sin embargo, a veces nos viene bien representarlo en un número concreto de bits (rellenando a ceros a la izquierda):
    - `format(5, 'b').zfill(8)`
    - `format(5, 'b').zfill(16)`
28. Supongamos que tenemos un valor de 8 bits representado en una variable de tipo *int*. Para simular esto podemos hacer lo siguiente:
    - `import random`
    - `i = random.randint(0,255)`
29. Para obtener por ejemplo el 2º bit (empezando por el bit de más peso; en la posición 0), no tendríamos más que hacer: `format(i, 'b').zfill(8)[2]`. Pruébalo.
30. Sin embargo, también podemos crear una máscara con todo ceros y un 1 en la posición 2 (por ejemplo, desplazando 5 posiciones a la izquierda un 1, es decir, `1 << 5`) y haciendo un AND lógico con `i`, conseguiríamos lo mismo. Si en la posición 2 (o en la 5, contando desde el bit de menos peso) había un 0, el resultado será 0 y si había un 1, el resultado será distinto de 0 (2^5). Prueba: `i & (1 << 5)`
31. Para concatenar los bits de varios bytes, lo más fácil es pasarlos primero a cadena de caracteres. Prueba:
    - `i1 = random.randint(0,255)`
    - `i2 = random.randint(0,255)`
    - `i1`
    - `i2`
    - `i12 = format(i1, 'b').zfill(8) + format(i2, 'b').zfill(8)`
    - `i12`
32. Si el resultado lo queremos como un entero, lo podemos obtener de distintas maneras, pasando por cadena de caracteres o no:
    - `int(i12, 2)`
    - `(i1 << 8) | i2`

Direcciones IP

33. Importa el paquete `socket` y obtén los 4 bytes asociados a la dirección IP 158.227.0.65 con la siguiente sentencia: `socket.inet_aton('158.227.0.65')`

Recuerda que los valores asociados a caracteres US-ASCII imprimibles Python los mostrará como caracteres (puedes usar el método *hex()*).

34. Para hacer la operación contraria y obtener la dirección IP en notación decimal con puntos: `socket.inet_ntoa(b'\x9e\xe3\x00\x41')`

~~Ejercicios~~

1. ~~Crea una cadena de bytes con el siguiente contenido (para representar números enteros en más de un byte usa la ordenación 'big'):~~
   - ~~Un campo de 16 bits con el número 7193.~~
   - ~~Un campo de 6 bits con el valor binario 011010.~~
   - ~~Un campo de 10 bits con el número 827.~~
   - ~~La cadena de carácteres 'ñopq' en codificación 'latin1'.~~
   - ~~Un byte con el número 0.~~
   - ~~La cadena de carácteres 'ñopq' en codificación 'utf-8'.~~
   - ~~Otro byte con el número 0.~~
   - ~~La dirección IP 127.0.0.53 en binario.~~
2. ~~Guarda en una variable de tipo cadena de bytes (*bytes*) el valor hexadecimal `3aa55690c1c9edf3fa006165696fc3bc00b90f3ac0` y encuentra cúal es el valor asociado a cada campo de los definidos a continuación (interpreta los números enteros representados en más de un byte con la ordenación 'big'):~~
   - ~~Un número entero en 16 bits.~~
   - ~~6 flags de un bit: f1, f2, f3, f4, f5 y f6.~~
   - ~~Un número entero en 10 bits.~~
   - ~~Una cadena de caracteres en codificación 'latin1' que termina con el byte '0'.~~
   - ~~Una cadena de caracteres en codificación 'utf-8' que termina con el byte '0'.~~
   - ~~Una dirección IP en binario.~~