Programación Avanzada IIC2233 2024-1

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Dante Pinto - Francisca Cattan

Anuncios

Jueves 23 de mayo 2023

- 1. Repasaremos dos temas:
 - a. Serialización
 - b. Excepciones
- Hoy tenemos la cuarta actividad, se entrega el lunes 27 a las 20:00 hrs.
- Solo para esta actividad responderemos issues abiertas hasta el viernes a las 20:00 hrs.
- 4. Se publicó un script que revisa elementos prohibidos para la T3.

Manejo de bytes

Manejo de bytes

I/O: Forma de interactuar con un programa.

En el contexto de archivos, todo archivo se guarda en un computador como *bytes*.

Un programa es capaz de leer y manipular directamente los bytes que representan un archivo.

Formato de almacenamiento de información de más bajo nivel.



- Entero entre 0 y 255 (2**8 1).
- Hexadecimal entre 0 y FF.
- Un literal (a, b, ...).

Tabla de conversión



Formato de almacenamiento de información de más bajo nivel.

En Python los bytes se representan con el objeto de tipo bytes.

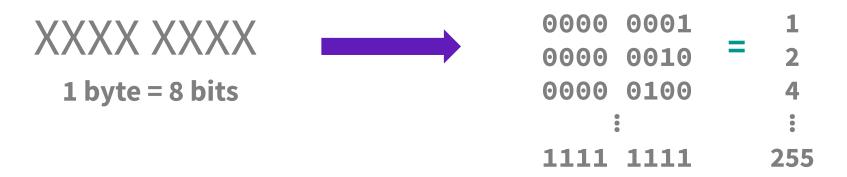
```
my_bytes = b"\x63\x6c\x69\x63\x68\xe9"
print(my_bytes) # b'clich\xe9'
```

```
XXXX XXXX 0000 0001
1 byte = 8 bits 0000 0100
:
1111 1111
```

Formato de almacenamiento de información de más bajo nivel.

En Python los bytes se representan con el objeto de tipo bytes.

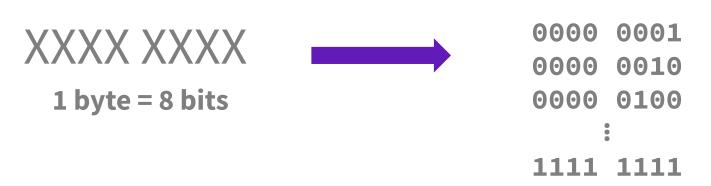
```
my_bytes = b"\x63\x6c\x69\x63\x68\xe9"
print(my_bytes) # b'clich\xe9'
```



Formato de almacenamiento de información de más bajo nivel.

En Python los bytes se representan con el objeto de tipo bytes.

```
my_bytes = b"\x63\x6c\x69\x63\x68\xe9"
print(my_bytes) # b'clich\xe9'
```



Caracteres ASCII

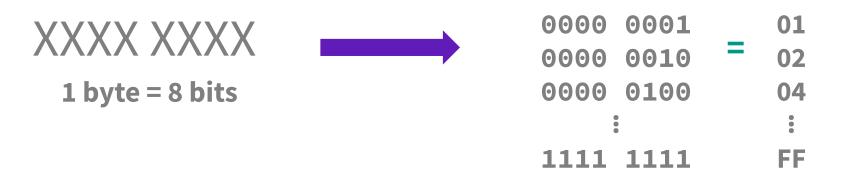
<u>Tabla ASCII</u>



Formato de almacenamiento de información de más bajo nivel.

En Python los bytes se representan con el objeto de tipo bytes.

```
my_bytes = b"\x63\x6c\x69\x63\x68\xe9"
print(my_bytes) # b'clich\xe9'
```



Bytearray

- Forma de hacer mutable nuestros bytes.
- Arreglos (listas) de bytes.

```
ba = bytearray(b' x15 xa3')
ba[0]
                                # 21
ba[1]
                                # 163
ba[0:1] = b' \x44'
                                # bytearray(b'\x44\xa3')
ba
len(ba)
                                # 2
max(ba)
                                # 163
ba[::-1]
                                # bytearray(b'\xa3\x44')
ba.zfill(4)
                                # bytearray(b'00\x44\xa3')
bytearray(b' \times 00 \times 00') + ba
                                # bytearray(b'\x00\x00\x44\xa3')
```

Bytearray

- Forma de hacer mutable nuestros bytes.
- Arreglos (listas) de bytes.

```
ba = bytearray(b'\x15\xa3')
ba[0]
ba[1]
ba[0:1] = b'\x44'
ba
len(ba)
max(ba)
ba[::-1]
ba.zfill(4)
bytearray(b'\x00\x00') + ba
```

```
Ojo que... ••

bytearray(b'0') != bytearray(b'\x00')

ord(bytearray(b'0')) => 48

ord(bytearray(b'\x00')) => 0

# bytearray(b'\xa3\x44')
# bytearray(b'00\x44\xa3')
# bytearray(b'\x00\x00\x44\xa3')
```

Serialización

Serialización de objetos

La serialización consiste en tener una **manera particular de guardar** *bytes*, de manera que estos puedan ser interpretados de manera inconfundible (por el mismo programa, otro programa o humanos).

En Python utilizamos dos módulos para hacer esto:

- pickle: Formato de Python, eficiente en almacenamiento, pero no es leíble y puede ser inseguro al deserializar.
- json: Formato interoperable y leíble, pero ineficiente en almacenamiento.

Con strings: dumps y loads

```
import pickle
                                   import json
tupla = ("a", 1, 3, "anya")
                                   tupla = ("a", 1, 3, "forger")
serializacion = pickle.dumps(tupla)
                                   serializacion = json.dumps(tupla)
print(serializacion)
                                   print(serializacion)
print(type(serializacion))
                                   print(type(serializacion))
print(pickle.loads(serializacion))
                                   print(json.loads(serializacion))
> <class 'bytes'>
                                   > <class 'str'>
                                   > ['a', 1, 3, 'forger']
> ('a', 1, 3, 'anya')
```

Con archivos: dump y load

```
import pickle
                                            import json
lista = [1, 2, 3, 7, 8, 3]
                                            lista = [1, 2, 3, 7, 8, 3]
with open("mi_lista.bin", 'wb') as file:
                                            with open("mi_lista.bin", 'w') as file:
    pickle.dump(lista, file)
                                                ison.dump(lista, file)
with open("mi_lista.bin", 'rb') as file:
                                            with open("mi_lista.bin", 'r') as file:
                                                lista_cargada = ison.load(file)
    lista_cargada = pickle.load(file)
print(f"¿Las listas son iguales?
                                            print(f"¿Las listas son iguales?
        {lista == lista_cargada}")
                                                    {lista == lista_cargada}")
print(f"¿Las listas son el mismo objeto?
                                            print(f"¿Las listas son el mismo objeto?
        {lista is lista_cargada}")
                                                    {lista is lista_cargada}")
> ¿Las listas son iguales? True
                                            > ¿Las listas son iguales? True
> ¿Las listas son el mismo objeto? False
                                            > ¿Las listas son el mismo objeto? False
```

Personalización en pickle: set y get state

```
class Persona:
    # ...
    def __getstate__(self):
        a_serializar = self.__dict__.copy()
        # Lo que retornemos será serializado por pickle
        return a_serializar

def __setstate__(self, state):
    # self.__dict__ contendrá los atributos deserializados
    self. dict = state
```

... y en json: JSONEncoder y object_hook

```
class PersonaEncoder(json.JSONEncoder):
    def default(self, obj):
                                         def hook_persona(diccionario):
        # Serializamos instancias
                                             # Recibe objetos de JSON
        diccionario = {
                                             # Podemos retornar lo que gueramos
            "nombre": obj.nombre,
                                             instancia = Persona(**diccionario)
            # . . .
                                             return instancia
        return diccionario
                                         json_string = ...
instancia = Persona(...)
                                         instancia = json.loads(
json_string = json.dumps(
                                             json_string,
    instancia,
                                             object_hook=hook_persona,
    cls=PersonaEncoder,
```

Excepciones

Mensajes de error

Hasta ahora nos hemos encontrado con mensajes de error al realizar ciertas operaciones no permitidas o utilizar métodos de forma incorrecta.

```
while True print("aqui vamos"):
print("no cierro comilla

^ > SyntaxError: invalid syntax

10 / 0
> ZeroDivisionError: division by zero.

► Error de sintaxis

► Error de sintaxis

► Error durante la ejecución
```

Excepciones Built-in

BaseException

SyntaxError

IndentationError

EOFError

NameError

ZeroDivisionError

IndexError

KeyError

AttributeError

TypeError

ValueError

Cada una tendrá una forma distinta de **capturar**, tratar y **manejar** la excepción.

<u>... y mas en la documentación</u>,



raise

Dada cierta condición, podríamos diseñar el **levantar** un tipo de excepción particular y añadir un mensaje adicional que informe al usuario sobre el error. Estas excepciones **interrumpen el flujo** del programa.

```
def verificar_largo(mensaje: bytearray) -> None:
    if len(mensaje) < 10:
        raise AttributeError("El largo del mensaje es menor a 10")
    return None

byte_arr = bytearray([1, 2, 3, 4, 5])
verificar_largo(byte_arr)

> AttributeError: El largo del mensaje es menor a 10

de excepción levantar?
```

raise

Dada cierta condición, podríamos diseñar el **levantar** un tipo de excepción particular y añadir un mensaje adicional que informe al usuario sobre el error. Estas excepciones **interrumpen el flujo** del programa.

```
def verificar_largo(mensaje: bytearray) -> None:
    if len(mensaje) < 10:
        raise AttributeError("El largo del mensaje es menor a 10")
    return None

byte_arr = bytearray([1, 2, 3, 4, 5])
verificar_largo(byte_arr)

> AttributeError: El largo del mensaje es menor a 10

¿Qué tipo de excepción
es más conveniente a
cada caso?
```

raise

Dada cierta condición, podríamos diseñar el **levantar** un tipo de excepción particular y añadir un mensaje adicional que informe al usuario sobre el error. Estas excepciones **interrumpen el flujo** del programa.

```
def verificar_largo(mensaje: bytearray) -> None:
    if len(mensaje) < 10:
        raise ValueError("El largo del mensaje es menor a 10")
    return None

byte_arr = bytearray([1, 2, 3, 4, 5])
verificar_largo(byte_arr)</pre>
Todo dependerá de tu
diseño
```

> AttributeError: El largo del mensaje es menor a 10

try y except

Si una excepción fue levantada durante la ejecución, podemos atraparla y manejarla. Lo que queremos **intentar** se encapsula dentro del bloque **try:** try: # Intentaremos leer un archivo JSON que no existe with open("archivo_json.json", "r") as json_file: data = json.load(json_file) except FileNotFoundError: # Atrapamos FileNotFoundError y levantamos otra excepcion raise ValueError ("Error al leer archivo JSON.") > FileNotFoundError: [Errno 2] No such file or directory: 'archivo_json.json' > During handling of the above exception, another exception occurred: > ValueError: Error al leer archivo JSON.

try y except

También podemos asignar la instancia del objeto error a una variable, y usar sus atributos o métodos. Dependiendo del manejo del error, **podemos continuar la ejecución del código.**

```
# Intentaremos leer un archivo JSON que no existe
    with open("archivo_json.json", "r") as json_file:
        data = json.load(json_file)

except FileNotFoundError as e:
    # Imprimimos un mensaje y el código continúa
    print(f"Error {e.__class__.__name__} al leer archivo JSON: {e.filename}")

print("...sigamos")

> Error FileNotFoundError al leer archivo JSON: archivo_json.json

> ...sigamos
```

Programación Avanzada IIC2233 2024-1

Hernán Valdivieso - Daniela Concha -

- Dante Pinto - Francisca Cattan