

# ***Programación Avanzada***

## **IIC2233 2023-2**

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Joaquín Tagle - Francisca Cattán



# Anuncios

Jueves 28 de septiembre 2023

1. Repasaremos dos temas:
    - a. Serialización
    - b. Excepciones
  2. Hoy tenemos la **cuarta** actividad, se entrega el miércoles 11 oct 2023 a las 20:00hrs.
  3. ¡Se viene la semana de receso!
  4. El 17 de octubre es el *midterm*
-

# 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 al nivel más mínimo de un archivo y manipularlo para generar más información.

---

# Bytes

Formato de almacenamiento de información de más bajo nivel.

**1 bit = valor 0 o 1**

XXXX XXXX

**1 byte = 8 bits**



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

[Tabla de conversión](#)



# Bytes

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'clích\xe9'
```

XXXX XXXX

1 byte = 8 bits



0000 0001

0000 0010

0000 0100

...

1111 1111

# Bytes

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'clích\xe9'
```

XXXX XXXX

**1 byte = 8 bits**



0000	0001	1
0000	0010	2
0000	0100	4
...		...
1111	1111	255

# Bytes

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

1 byte = 8 bits



0000 0001  
0000 0010  
0000 0100  
...  
1111 1111

Caracteres  
ASCII

[Tabla ASCII](#)





# Bytes

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

1 byte = 8 bits



0000	0001	01
0000	0010	02
0000	0100	04
...		...
1111	1111	FF

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

```
ba.zfill(4) # bytearray(b'00\x44\xa3')
bytearray(b'\x00\x00') + ba # bytearray(b'\x00\x00\x44\xa3')
```

👁👁 ojo que

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

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

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

# 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.

# Cómo se ven pickle y JSON

```
b'\x80\x03}q\x00(X\n\x00\x00\x00first_nameq\x01X\x06\x00\x00\x00Alexisq\x02X\t\x00\x00\x00last_nameq\x03X\x08\x00\x00\x00S\xc3\xa1nchezq\x04X\t\x00\x00\x00birthdateq\x05X\n\x00\x00\x001988-12-19q\x06X\x06\x00\x00\x00heightq\x07G?\xfb\n=p\xa3\xd7\nX\x04\x00\x00\x00clubq\x08}q\t(X\x04\x00\x00\x00nameq\nX\x0f\x00\x00\x00Inter de MIL\xc3\xa1nq\x0bX\x07\x00\x00\x00foundedq\x0cMt\x07uX\x07\x00\x00\x00aliasesq\r]q\x0e(X\x0f\x00\x00\x00Ni\xc3\xb1oMaravillaq\x0fX\x15\x00\x00\x00La Ardilla de Atacamaq\x10eX\n\x00\x00\x00girlfriendq\x11NX\x07\x00\x00\x00injuredq\x12\x88u.'
```

```
{
  "first_name": "Alexis",
  "last_name": "Sánchez",
  "birthdate": "1988-12-19",
  "height": 1.69,
  "club": {
    "name": "Inter de MILán",
    "founded": 1908
  },
  "aliases": [
    "Niño Maravilla",
    "La Ardilla de Atacama"
  ],
  "girlfriend": null,
  "injured": true
}
```

# Con *strings*: dumps y loads

```
import pickle
```

```
tupla = ("a", 1, 3, "hola")  
serializacion = pickle.dumps(tupla)
```

```
print(serializacion)  
print(type(serializacion))  
print(pickle.loads(serializacion))
```

```
> b'\x80\x03(X\x01\x00\x00\x00aq[...]'  
> <class 'bytes'>  
> ('a', 1, 3, 'hola')
```

```
import json
```

```
tupla = ("a", 1, 3, "hola")  
serializacion = json.dumps(tupla)
```

```
print(serializacion)  
print(type(serializacion))  
print(json.loads(serializacion))
```

```
> ["a", 1, 3, "hola"]  
> <class 'str'>  
> ['a', 1, 3, 'hola']
```

# Con archivos: dump y load

```
import pickle
```

```
lista = [1, 2, 3, 7, 8, 3]
```

```
with open("mi_lista.bin", 'wb') as file:  
    pickle.dump(lista, file)
```

```
with open("mi_lista.bin", 'rb') as file:  
    lista_cargada = pickle.load(file)
```

```
print(f"¿Las listas son iguales? {lista ==  
lista_cargada}")
```

```
print(f"¿Las listas son el mismo objeto?  
{lista is lista_cargada}")
```

```
> ¿Las listas son iguales? True  
> ¿Las listas son el mismo objeto? False
```

```
import json
```

```
lista = [1, 2, 3, 7, 8, 3]
```

```
with open("mi_lista.bin", 'w') as file:  
    json.dump(lista, file)
```

```
with open("mi_lista.bin", 'r') as file:  
    lista_cargada = json.load(file)
```

```
print(f"¿Las listas son iguales? {lista ==  
lista_cargada}")
```

```
print(f"¿Las listas son el mismo objeto?  
{lista is lista_cargada}")
```

```
> ¿Las listas son iguales? True  
> ¿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):  
        # Serializamos instancias  
        diccionario = {  
            "nombre": obj.nombre,  
            # ...  
        }  
        return diccionario
```

```
instancia = Persona(...)  
json_string = json.dumps(  
    instancia,  
    cls=PersonaEncoder,  
)
```

```
def hook_persona(diccionario):  
    # Recibe objetos de JSON  
    # Podemos retornar lo que queramos  
    instancia = Persona(**diccionario)  
    return instancia
```

```
json_string = ...  
instancia = json.loads(  
    json_string,  
    object_hook=hook_persona,  
)
```

# 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"):  
while print("aqui vamos"):  
    ^
```

```
SyntaxError: invalid syntax
```

→ **Error de sintaxis**

```
10 / 0
```

```
ZeroDivisionError: division by zero.
```

→ **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.

... y mas en la documentación



# 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)
```

¿Es adecuado?

¿Cómo decido qué tipo  
de excepción levantar?

```
> AttributeError: El largo del mensaje es menor a 10
```

# 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)
```

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

```
> AttributeError: El largo del mensaje es menor a 10
```

# 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)
```

Todo dependerá de tu diseño

```
> ValueError: 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.**

```
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 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 2023-2**

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Joaquín Tagle - Francisca Cattán



# Comentarios Actividad 4

> Tomen agua