ASICI

Asociación Internacional de Ciberseguridad

Ejemplos de Python nivel Avanzado

1. Metaclases en Python

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creando la clase {name}")
        return super().__new__(cls, name, bases, dct)
class MiClase(metaclass=Meta):
    pass
obj = MiClase()
```

Explicación: Una metaclase es una clase de clases, es decir, controla la creación de clases en Python. En este ejemplo, `Meta` es una metaclase que sobrescribe el método especial `__new__()` para interceptar la creación de la clase `MiClase`. Cuando se crea una instancia de `MiClase`, se imprime un mensaje. Las metaclases permiten personalizar la forma en que se definen las clases.

2. Decorador con parámetros

```
def decorador_con_parametros(texto):
    def decorador(func):
        def wrapper(*args, **kwargs):
            print(f"Texto: {texto}")
            return func(*args, **kwargs)
        return wrapper
    return decorador
@decorador_con_parametros("Ejemplo avanzado")
def saludar(nombre):
    print(f"Hola, {nombre}")
saludar("Juan")
```

Explicación: Este decorador permite pasar parámetros al decorador mismo. En este caso, el parámetro `texto` se inyecta en el decorador. El decorador `decorador_con_parametros()` toma una función `func` como entrada, y el `wrapper` la envuelve con un comportamiento adicional que imprime el texto proporcionado antes de ejecutar la función decorada.

3. Uso de `@property`

```
class Persona:
    def __init__(self, nombre):
        self._nombre = nombre
    @property
    def nombre(self):
        return self._nombre
    @nombre.setter
    def nombre(self, valor):
        self._nombre = valor
persona = Persona("Ana")
persona.nombre = "Juan"
print(persona.nombre)
```

Explicación: El decorador `@property` permite definir métodos que se comportan como atributos. El método `nombre()` actúa como un getter, mientras que el decorador `@nombre.setter` permite modificar el valor del atributo privado `_nombre`. Así, `nombre` se puede acceder y modificar como si fuera un atributo, pero sigue estando controlado por métodos.

4. Contexto personalizado con `contextlib`

```
from contextlib import contextmanager
@contextmanager
def recurso():
    print("Entrando al contexto")
    yield
    print("Saliendo del contexto")
with recurso():
    print("Dentro del contexto")
```

Explicación: `contextlib.contextmanager` permite crear contextos personalizados utilizando el decorador `@contextmanager`. En lugar de definir manualmente `__enter__` y `__exit__` como en un contexto tradicional, se usa `yield` para indicar el punto donde se ejecuta el código dentro del bloque `with`. Esto es útil para manejar recursos como archivos o conexiones.

5. Memoización

```
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper
@memoize
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
print(fibonacci(35))
```

Explicación: La memoización es una técnica de optimización que almacena los resultados de una función para evitar cálculos repetidos. Aquí, el decorador `memoize` guarda los resultados de `fibonacci()` en un diccionario `cache`, lo que acelera el cálculo al evitar volver a calcular los valores ya vistos.

6. Programación funcional con `reduce`

```
from functools import reduce
numeros = [1, 2, 3, 4, 5]
resultado = reduce(lambda x, y: x * y, numeros)
print(resultado)
```

Explicación: `reduce()` toma una función y una secuencia, aplicando la función acumulativamente a los elementos. En este caso, la función `lambda` multiplica los elementos de la lista `numeros` de izquierda a derecha. El resultado final es el producto de todos los elementos de la lista.

7. Uso de `asyncio` para concurrencia asíncrona

```
import asyncio
async def tarea(nombre, tiempo):
    print(f"Inicio de {nombre}")
    await asyncio.sleep(tiempo)
```

```
    print(f"Fin de {nombre}")
async def main():
    await asyncio.gather(tarea("Tarea 1", 2), tarea("Tarea 2", 1))
asyncio.run(main())
```

Explicación: `asyncio` permite ejecutar múltiples tareas de forma asíncrona, lo que es útil para operaciones que involucran I/O, como acceder a archivos o realizar llamadas de red. En este ejemplo, `asyncio.gather()` ejecuta dos tareas asíncronas en paralelo. El uso de `await` suspende la tarea para permitir que otras tareas se ejecuten mientras espera.

8. Acceso concurrente con `threading`

```
import threading
contador = 0
lock = threading.Lock()

def incrementar():
    global contador
    for _ in range(100000):
        with lock:
            contador += 1
hilos = [threading.Thread(target=incrementar) for _ in range(2)]
for hilo in hilos:
    hilo.start()
for hilo in hilos:
    hilo.join()
print(f"Contador final: {contador}")
```

Explicación: El módulo `threading` permite la ejecución de múltiples hilos de manera concurrente. Para evitar que múltiples hilos accedan y modifiquen variables al mismo tiempo, se utiliza un bloqueo (`Lock`). En este ejemplo, dos hilos incrementan un contador de manera segura utilizando el bloqueo para evitar condiciones de carrera.

9. Acceso concurrente con `multiprocessing`

```
import multiprocessing
def tarea(n):
    return n * n
```

```
with multiprocessing.Pool(4) as pool:
    resultado = pool.map(tarea, [1, 2, 3, 4])
    print(resultado)
```

Explicación: `multiprocessing` permite la ejecución de múltiples procesos en paralelo, lo cual es útil para tareas intensivas en CPU. En este ejemplo, se utiliza un `Pool` de 4 procesos para realizar cálculos de forma concurrente. Cada proceso calcula el cuadrado de un número en paralelo.

10. Algoritmo de búsqueda binaria

```
def busqueda_binaria(arr, objetivo):
    izquierda, derecha = 0, len(arr) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if arr[medio] == objetivo:
            return medio
        elif arr[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1
arr = [1, 3, 5, 7, 9]
print(busqueda_binaria(arr, 5))
```

Explicación: La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada. El algoritmo divide la lista repetidamente por la mitad, comparando el elemento medio con el valor buscado. En este ejemplo, se busca el número `5` en la lista `arr`.

11. Algoritmo de ordenamiento rápido (Quicksort)

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivote = arr[len(arr) // 2]
    izquierda = [x for x in arr if x < pivote]
    medio = [x for x in arr if x == pivote]
    derecha = [x for x in arr if x > pivote]
    return quicksort(izquierda) + medio + quicksort(derecha)
```

```
arr = [3, 6, 8, 10, 1, 2, 1]
print(quicksort(arr))
```

Explicación: Quicksort es un algoritmo de ordenamiento que selecciona un pivote y divide la lista en dos sublistas: una con elementos menores que el pivote y otra con elementos mayores. Luego ordena cada sublista recursivamente. Es eficiente en la práctica y tiene un rendimiento promedio de O(nlogn)O(n \log n)O(nlogn).

12. Manejo de bases de datos con SQLite

```
import sqlite3
conn = sqlite3.connect(':memory:')
cursor = conn.cursor()
cursor.execute('''CREATE TABLE usuarios (nombre text, edad
integer)''')
cursor.execute('''INSERT INTO usuarios VALUES ('Ana', 30)''')
conn.commit()
cursor.execute('SELECT * FROM usuarios')
print(cursor.fetchall())
conn.close()
```

Explicación: Este código utiliza SQLite, una base de datos ligera integrada en Python. Primero, se crea una tabla `usuarios`, luego se inserta un registro en la tabla y se realiza una consulta para recuperar todos los registros. La conexión a la base de datos se realiza en memoria (`:memory:`), lo que significa que los datos se almacenan temporalmente.

13. Creación de una API con Flask

```
from flask import Flask, jsonify
app = Flask(__name__)
@app.route('/')
def index():
    return jsonify({"mensaje": "Hola, mundo"})
if __name__ == '__main__':
    app.run(debug=True)
```

Explicación: Este código crea una API web simple usando Flask. Define una ruta (`/`) que devuelve un mensaje en formato JSON. Flask es un framework ligero para construir aplicaciones web. Al ejecutar el script, se inicia un servidor local que escucha en el puerto 5000 por defecto.

14. Uso avanzado de `dataclasses`

```
from dataclasses import dataclass
@dataclass
class Persona:
    nombre: str
    edad: int
    activo: bool = True
persona = Persona(nombre="Juan", edad=28)
print(persona)
```

Explicación: Las `dataclasses` permiten definir clases que se utilizan principalmente para almacenar datos sin tener que escribir código adicional. El decorador `@dataclass` genera automáticamente métodos como `__init__`, `__repr__` y `__eq__` basándose en los atributos definidos. Aquí se define una clase `Persona` con tres atributos y se crea una instancia de la misma.

15. Manejo de eventos con `asyncio` y `queue`

```
import asyncio
import random
async def productor(queue):
    for _ in range(10):
        item = random.randint(1, 100)
        await queue.put(item)
        print(f"Producido {item}")
        await asyncio.sleep(1)
async def consumidor(queue):
    for _ in range(10):
        item = await queue.get()
        print(f"Consumido {item}")
        queue.task_done()
async def main():
    queue = asyncio.Queue()
```

```
    productor_task = asyncio.create_task(productor(queue))
    consumidor_task = asyncio.create_task(consumidor(queue))
    await asyncio.gather(productor_task, consumidor_task)

asyncio.run(main())
```

Explicación: Este código utiliza `asyncio` para implementar un patrón productor-consumidor asíncrono. Un `productor` genera elementos y los pone en una cola (`queue`), mientras que un `consumidor` extrae esos elementos y los procesa. El `await queue.put()` y `await queue.get()` permiten manejar las operaciones de manera asíncrona, permitiendo que el productor y el consumidor funcionen simultáneamente sin bloquearse mutuamente.

16. Web scraping con `BeautifulSoup`

```
import requests
from bs4 import BeautifulSoup
response = requests.get('https://www.python.org')
soup = BeautifulSoup(response.content, 'html.parser')
for enlace in soup.find_all('a'):
    print(enlace.get('href'))
```

Explicación: `BeautifulSoup` es una biblioteca que permite analizar (parsear) contenido HTML. Este ejemplo obtiene el contenido de la página web de Python usando `requests`, luego usa `BeautifulSoup` para encontrar todos los enlaces (`<a>`) en la página y los imprime. Es una técnica común en web scraping para extraer datos de páginas web.

17. Pipeline de Machine Learning con `scikit-learn`

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data.data,
data.target, test_size=0.2)
clf = RandomForestClassifier()
clf.fit(X_train, y_train)
```

```
predicciones = clf.predict(X_test)
print(f"Exactitud: {accuracy_score(y_test, predicciones)}")
```

Explicación: Este código utiliza `scikit-learn`, una biblioteca popular para machine learning. Carga el conjunto de datos de iris, divide los datos en conjuntos de entrenamiento y prueba, entrena un modelo de Random Forest y evalúa su exactitud en los datos de prueba. Este pipeline es una estructura común en tareas de aprendizaje automático.

18. Implementación de una cola utilizando dos pilas

```
class ColaConPilas:
    def __init__(self):
        self.pila1 = []
        self.pila2 = []
    def encolar(self, valor):
        self.pila1.append(valor)
    def desencolar(self):
        if not self.pila2:
            while self.pila1:
                self.pila2.append(self.pila1.pop())
        return self.pila2.pop()
cola = ColaConPilas()
cola.encolar(1)
cola.encolar(2)
print(cola.desencolar())  # Imprime 1
```

Explicación: Este código implementa una cola (FIFO) utilizando dos pilas (LIFO). Los elementos se encolan en `pila1` y, cuando se necesita desencolar, se pasan de `pila1` a `pila2`, invirtiendo su orden, de modo que el primer elemento en `pila1` sea el primero en salir de `pila2`. Esto simula el comportamiento de una cola usando solo pilas.

19. Lectura y escritura en archivos CSV con `pandas`

```
import pandas as pd
data = {'Nombre': ['Juan', 'Ana'], 'Edad': [28, 25]}
df = pd.DataFrame(data)
df.to_csv('archivo.csv', index=False)
df_cargado = pd.read_csv('archivo.csv')
print(df_cargado)
```

Explicación: Este código muestra cómo trabajar con archivos CSV utilizando `pandas`. Primero, se crea un DataFrame y se guarda en un archivo CSV. Luego, se carga el archivo CSV en un nuevo DataFrame y se imprime. `pandas` facilita el manejo de datos tabulares y el intercambio con otros formatos como CSV.

20. Hacer peticiones HTTP con `requests`

```
import requests
response = requests.get('https://api.github.com')
print(response.json())
```

Explicación: `requests` es una biblioteca simple pero poderosa para hacer peticiones HTTP. Este ejemplo realiza una petición GET a la API de GitHub y utiliza el método `json()` para convertir la respuesta en formato JSON a un diccionario de Python.

21. Acceso a una API con autenticación OAuth

```
import requests
from requests_oauthlib import OAuth1
auth = OAuth1('TU_CONSUMER_KEY', 'TU_CONSUMER_SECRET',
'TU_ACCESS_TOKEN', 'TU_ACCESS_SECRET')
response =
requests.get('https://api.twitter.com/1.1/account/verify_credentials.j
son', auth=auth)
print(response.json())
```

Explicación: Este código utiliza `requests_oauthlib` para realizar una petición autenticada a la API de Twitter utilizando el protocolo OAuth 1.0a. Los tokens de acceso (`ACCESS_TOKEN` y `ACCESS_SECRET`) permiten autenticar al usuario y acceder a sus datos. En este caso, la API verifica las credenciales del usuario.

22. Simulaciones Monte Carlo

```
import random
def estimar_pi(n):
    adentro_circulo = 0
    for _ in range(n):
        x, y = random.random(), random.random()
        if x**2 + y**2 <= 1:
            adentro_circulo += 1
```

```
    return (adentro_circulo / n) * 4
print(estimar_pi(100000))
```

Explicación: Este código utiliza un método de simulación de Monte Carlo para estimar el valor de π. Genera puntos aleatorios dentro de un cuadrado unitario y calcula cuántos puntos caen dentro del círculo inscrito. La razón entre los puntos dentro del círculo y el total de puntos se multiplica por 4 para obtener una aproximación de π.

23. Creación de clases dinámicamente

```
def crear_clase(nombre, atributos):
    return type(nombre, (object,), atributos)
Persona = crear_clase('Persona', {'nombre': 'Juan', 'edad': 28})
persona = Persona()
print(f"{persona.nombre}, {persona.edad}")
```

Explicación: Este código usa la función `type()` para crear clases dinámicamente. `type()` es una metaclase que toma el nombre de la clase, una tupla de clases base, y un diccionario de atributos para crear una nueva clase. Aquí, se crea una clase `Persona` con los atributos `nombre` y `edad` en tiempo de ejecución.

24. Subprocesos con `concurrent.futures`

```
from concurrent.futures import ThreadPoolExecutor
def tarea(n):
    return n * n
with ThreadPoolExecutor(max_workers=4) as executor:
    resultados = list(executor.map(tarea, range(10)))
    print(resultados)
```

Explicación: `concurrent.futures` facilita la ejecución de tareas en paralelo. Aquí, se utiliza `ThreadPoolExecutor` para crear un pool de hilos que ejecuta la función `tarea()` en paralelo. `executor.map()` aplica la función a los elementos de la secuencia `range(10)` y devuelve los resultados en paralelo.

25. Procesamiento de imágenes con `PIL` (Pillow)

```
from PIL import Image
imagen = Image.open('imagen.jpg')
imagen.thumbnail((100, 100))
```

```
imagen.save('thumbnail.jpg')
```

Explicación:

Este ejemplo utiliza Pillow (PIL), una biblioteca popular para el procesamiento de imágenes en Python. Aquí, el programa abre una imagen llamada `imagen.jpg`, la redimensiona para crear una miniatura de 100x100 píxeles usando el método `thumbnail()`, y luego guarda esta miniatura en un archivo llamado `thumbnail.jpg`. Pillow es muy útil para operaciones como redimensionar, convertir, rotar y aplicar filtros a imágenes.

- **`Image.open()`: Abre una imagen y la prepara para ser procesada.**
- **`thumbnail()`: Cambia el tamaño de la imagen manteniendo la relación de aspecto original, sin distorsionar la imagen.**
- **`save()`: Guarda la imagen procesada en un archivo.**

26. Análisis de texto con `spaCy`

```
import spacy
nlp = spacy.load('en_core_web_sm')
doc = nlp("Python is a popular programming language.")
for token in doc:
    print(token.text, token.pos_)
```

Explicación:

`spaCy` es una poderosa biblioteca de procesamiento del lenguaje natural (NLP). Este ejemplo carga un modelo de lenguaje preentrenado (`en_core_web_sm`) para el inglés y luego procesa una oración. El texto se divide en "tokens" (palabras y signos de puntuación), y para cada token, se imprime el texto y la categoría gramatical (parte de la oración, o POS, por sus siglas en inglés).

- **`nlp()`: Convierte el texto en un objeto `Doc` que contiene los tokens, relaciones gramaticales, entidades, etc.**
- **`token.pos_`: Identifica la parte del discurso (sustantivo, verbo, adjetivo, etc.) del token.**

27. Trabajar con `deque` para listas doblemente enlazadas

```
from collections import deque
cola = deque([1, 2, 3])
```

```
cola.append(4)
cola.appendleft(0)
print(cola)
cola.pop()
cola.popleft()
print(cola)
```

Explicación:

`deque` **(double-ended queue) es una estructura de datos altamente eficiente para insertar y eliminar elementos desde ambos extremos. Es útil cuando necesitas una lista donde las operaciones de agregar o quitar elementos desde el principio o el final deben ser rápidas.**

- `append()`**: Agrega un elemento al final de la** `deque`**.**
- `appendleft()`**: Agrega un elemento al principio de la** `deque`**.**
- `pop()`**: Elimina el último elemento.**
- `popleft()`**: Elimina el primer elemento.**

Este ejemplo muestra cómo trabajar con una `deque`**, agregando elementos por ambos extremos y luego eliminándolos.**

28. Uso de `NamedTuple`

```
from collections import namedtuple
Persona = namedtuple('Persona', 'nombre edad')
persona = Persona(nombre="Ana", edad=30)
print(persona.nombre, persona.edad)
```

Explicación:

`namedtuple` **es una estructura de datos similar a las tuplas normales, pero con la ventaja de permitir acceder a sus elementos por nombre en lugar de por índice. Este ejemplo crea una tupla con nombre llamada** `Persona`**, con los campos** `nombre` **y** `edad`**. Luego se crea una instancia de** `Persona` **y se acceden a los campos como si fueran atributos de una clase.**

- `namedtuple()`**: Define una tupla con nombre, especificando los campos.**
- `persona.nombre` **y** `persona.edad`**: Permiten acceder a los valores de los campos por nombre.**

29. *Algoritmo de búsqueda A (A-Star)**

```
from queue import PriorityQueue
def a_star(inicio, meta, grafo, heuristica):
    cola = PriorityQueue()
    cola.put((0, inicio))
    costos = {inicio: 0}
    came_from = {inicio: None}
    while not cola.empty():
        _, actual = cola.get()
        if actual == meta:
            return reconstruir_camino(came_from, actual)
        for vecino in grafo[actual]:
            nuevo_costo = costos[actual] + grafo[actual][vecino]
            if vecino not in costos or nuevo_costo < costos[vecino]:
                costos[vecino] = nuevo_costo
                prioridad = nuevo_costo + heuristica[vecino]
                cola.put((prioridad, vecino))
                came_from[vecino] = actual
def reconstruir_camino(came_from, actual):
    camino = []
    while actual is not None:
        camino.append(actual)
        actual = came_from[actual]
    return camino[::-1]
grafo = {
    'A': {'B': 1, 'C': 3},
    'B': {'D': 1},
    'C': {'D': 1},
    'D': {}
}
heuristica = {'A': 4, 'B': 2, 'C': 2, 'D': 0}
print(a_star('A', 'D', grafo, heuristica))
```

Explicación:

El algoritmo A* (A-Star) es un algoritmo de búsqueda en grafos que se utiliza para encontrar la ruta más corta entre dos puntos. Usa una combinación de búsqueda de costo mínimo y una heurística (estimación de distancia) para priorizar los caminos.

- `PriorityQueue`: **La cola de prioridad permite extraer siempre el elemento con el menor "costo" (en este caso, la combinación del costo recorrido y la heurística).**
- `a_star()`: **Implementa el algoritmo A*. Empieza desde el nodo inicial e intenta encontrar la ruta más corta al nodo objetivo, utilizando la heurística para decidir el siguiente nodo a visitar.**
- `reconstruir_camino()`: **Una vez que el objetivo se alcanza, esta función reconstruye la ruta desde el nodo inicial hasta el objetivo.**

30. Algoritmo de Backtracking (Problema de las N-Reinas)

```
def es_seguro(tablero, fila, col):
    for i in range(col):
        if tablero[fila][i] == 1:
            return False
    for i, j in zip(range(fila, -1, -1), range(col, -1, -1)):
        if tablero[i][j] == 1:
            return False
    for i, j in zip(range(fila, len(tablero)), range(col, -1, -1)):
        if tablero[i][j] == 1:
            return False
    return True
def resolver_n_reinas(tablero, col):
    if col >= len(tablero):
        return True
    for i in range(len(tablero)):
        if es_seguro(tablero, i, col):
            tablero[i][col] = 1
            if resolver_n_reinas(tablero, col + 1):
                return True
            tablero[i][col] = 0
    return False
def imprimir_tablero(tablero):
    for fila in tablero:
        print(fila)
n = 4
tablero = [[0] * n for _ in range(n)]
if resolver_n_reinas(tablero, 0):
    imprimir_tablero(tablero)
else:
```

```
    print("No tiene solución.")
```

Explicación:

Este código resuelve el problema de las N-Reinas usando backtracking, una técnica de prueba y error que deshace las decisiones cuando se detecta que una solución parcial no puede llevar a una solución completa.

- `es_seguro()`**: Verifica si es seguro colocar una reina en una celda específica del tablero. Se asegura de que no haya otras reinas en la misma fila, columna o diagonales.**
- `resolver_n_reinas()`**: Utiliza recursión para intentar colocar una reina en cada columna. Si una colocación es válida, procede a la siguiente columna. Si no se puede colocar una reina, retrocede (backtracking) y prueba otras posiciones.**
- `imprimir_tablero()`**: Imprime el estado actual del tablero.**

Este es un problema clásico de backtracking que se resuelve buscando de forma eficiente configuraciones válidas en un tablero de ajedrez.