ASICI

Asociación Internacional de Ciberseguridad

Ejemplos de Python nivel Intermedio

1. Clase básica en Python

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad}
años.")

persona = Persona("Ana", 30)
persona.saludar()
```

Este código define una clase llamada `Persona` que tiene un constructor (`__init__`) con dos atributos: `nombre` y `edad`. La clase también tiene un método `saludar()` que imprime un mensaje con el nombre y la edad de la persona. Luego, se crea una instancia de la clase y se llama al método `saludar()`.

2. Herencia

```
class Animal:
    def hacer_sonido(self):
        print("El animal hace un sonido.")

class Perro(Animal):
    def hacer_sonido(self):
        print("El perro ladra.")

perro = Perro()
perro.hacer_sonido()
```

En este ejemplo, `Perro` hereda de la clase `Animal`. Ambas clases tienen un método `hacer_sonido()`, pero `Perro` lo sobrescribe (polimorfismo). Se crea una instancia de `Perro` y se llama al método `hacer_sonido()` de la subclase.

3. Decoradores en Python

```
def decorador(func):
    def wrapper():
        print("Función decorada")
        func()
        print("Finaliza la función decorada")
    return wrapper

@decorador
def saludar():
    print("Hola Mundo")

saludar()
```

Este código define un decorador que modifica el comportamiento de la función `saludar()`. El decorador añade comportamiento antes y después de la ejecución de `saludar()`, imprimiendo mensajes adicionales. Los decoradores son útiles para modificar funciones sin cambiar su definición original.

4. Manejo avanzado de excepciones

```
try:
    with open('archivo_inexistente.txt', 'r') as archivo:
        contenido = archivo.read()
except FileNotFoundError as e:
    print(f"Error: {e}")
finally:
    print("Esto se ejecuta siempre.")
```

Este código intenta abrir un archivo inexistente, lo que provoca un error `FileNotFoundError`. El error es capturado por el bloque `except` y se imprime un mensaje. El bloque `finally` se ejecuta siempre, independientemente de si ocurrió un error o no.

5. Generadores

```
def generador_cuadrados(limite):
    for i in range(limite):
        yield i ** 2

for numero in generador_cuadrados(5):
    print(numero)
```

Los generadores son funciones que devuelven elementos uno a uno usando la palabra clave `yield`. Este generador devuelve los cuadrados de los números del 0 al 4. Cada vez que el generador se itera, devuelve el siguiente valor sin almacenar toda la secuencia en memoria.

6. Comprensión de diccionarios

```
numeros = [1, 2, 3, 4]
cuadrados = {x: x**2 for x in numeros}
print(cuadrados)
```

Este código utiliza una comprensión de diccionario para crear un diccionario donde las claves son los números de la lista `numeros`, y los valores son sus cuadrados. La comprensión de diccionarios es una forma compacta de crear diccionarios.

7. Expresiones regulares

```
import re
patron = r"\d+"
texto = "Hay 123 números en esta cadena."
coincidencias = re.findall(patron, texto)
print(coincidencias)
```

Este ejemplo utiliza la biblioteca `re` (expresiones regulares) para encontrar todos los números en una cadena de texto. El patrón `\d+` busca uno o más dígitos. `re.findall()` devuelve una lista con todas las coincidencias encontradas.

8. Lectura y escritura en formato JSON

```
import json
data = {"nombre": "Juan", "edad": 25}
```

```
json_data = json.dumps(data)
print(json_data)
data_cargada = json.loads(json_data)
print(data_cargada["nombre"])
```

Este código utiliza la biblioteca `json` para convertir un diccionario Python en una cadena JSON con `json.dumps()`, y luego vuelve a convertirla a un diccionario usando `json.loads()`. JSON es un formato común para intercambiar datos.

9. Operaciones con matrices usando NumPy

```
import numpy as np
matriz = np.array([[1, 2], [3, 4]])
transpuesta = matriz.T
print(transpuesta)
```

Este ejemplo utiliza la biblioteca `NumPy` para crear una matriz y luego calcula su transpuesta usando el atributo `.T`. `NumPy` es ampliamente usado en matemáticas, ciencia de datos y procesamiento numérico.

10. Pandas para manejo de datos

```
import pandas as pd
data = {'Nombre': ['Ana', 'Juan', 'Pedro'], 'Edad': [25, 30, 22]}
df = pd.DataFrame(data)
print(df)
```

Este código crea un DataFrame de `pandas`, que es una estructura de datos similar a una tabla. `pandas` es una biblioteca poderosa para la manipulación y análisis de datos.

11. Crear un contexto personalizado con `with`

```
class ContextoPersonalizado:
    def __enter__(self):
        print("Entrando al contexto")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
```

```
        print("Saliendo del contexto")

with ContextoPersonalizado():
    print("Dentro del contexto")
```

Este código define un contexto personalizado usando las funciones `__enter__()` y `__exit__()` que controlan lo que ocurre al entrar y salir del contexto `with`. Esto es útil para gestionar recursos, como abrir y cerrar archivos.

12. Uso de `itertools`

```
import itertools

data = [1, 2, 3]
combinaciones = list(itertools.permutations(data))
print(combinaciones)
```

`itertools` es una biblioteca que proporciona herramientas avanzadas para trabajar con iteradores. En este ejemplo, `itertools.permutations()` genera todas las permutaciones posibles de la lista `data`.

13. Uso de `functools.reduce`

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

`reduce()` aplica una función acumulativa (en este caso, una suma) a los elementos de una lista. Aquí, se suman todos los números de la lista utilizando una función `lambda`.

14. Clases abstractas

```
from abc import ABC, abstractmethod

class Forma(ABC):
    @abstractmethod
    def area(self):
        pass
```

```
class Cuadrado(Forma):
    def __init__(self, lado):
        self.lado = lado

    def area(self):
        return self.lado ** 2

cuadrado = Cuadrado(4)
print(cuadrado.area())
```

Este ejemplo utiliza la clase `ABC` (Abstract Base Class) para definir una clase abstracta `Forma` con un método abstracto `area()`. Las clases abstractas no pueden ser instanciadas directamente, pero obligan a las subclases a implementar ciertos métodos.

15. Métodos estáticos y de clase

```
class Matematica:
    @staticmethod
    def suma(a, b):
        return a + b
    @classmethod
    def mensaje(cls):
        return "Esto es un mensaje de la clase Matematica"
print(Matematica.suma(3, 4))
print(Matematica.mensaje())
```

Este código muestra cómo definir un método estático (`@staticmethod`) y un método de clase (`@classmethod`). Los métodos estáticos no dependen de una instancia de clase, mientras que los métodos de clase pueden acceder a la clase en sí (`cls`).

16. Usar `map()`

```
numeros = [1, 2, 3, 4]
cuadrados = list(map(lambda x: x**2, numeros))
print(cuadrados)
```

`map()` aplica una función (en este caso, una función `lambda` que calcula el cuadrado) a cada elemento de una lista. El resultado es un nuevo iterador que luego se convierte en una lista.

17. Serialización con `pickle`

```
import pickle

data = {'nombre': 'Ana', 'edad': 25}
with open('data.pkl', 'wb') as archivo:
    pickle.dump(data, archivo)
with open('data.pkl', 'rb') as archivo:
    data_cargada = pickle.load(archivo)
    print(data_cargada)
```

Este código usa la biblioteca `pickle` para serializar (guardar) un objeto Python en un archivo binario (`data.pkl`) y luego deserializar (cargar) ese objeto de vuelta a su estado original.

18. Crear un archivo ZIP

```
import zipfile
with zipfile.ZipFile('archivo.zip', 'w') as zipf:
    zipf.write('archivo.txt')
```

Este ejemplo utiliza la biblioteca `zipfile` para crear un archivo ZIP (`archivo.zip`) y agregar el archivo `archivo.txt` dentro de él.

19. Tiempos de ejecución con `timeit`

```
import timeit
codigo = """
numeros = [x for x in range(100)]
"""
tiempo = timeit.timeit(codigo, number=1000)
print(tiempo)
```

`timeit` mide el tiempo que toma ejecutar un código específico. En este caso, mide cuánto tiempo toma ejecutar el código que genera una lista 1000 veces.

20. Uso de `Counter` para contar elementos en una lista

```
from collections import Counter
lista = [1, 2, 2, 3, 3, 3]
frecuencia = Counter(lista)
print(frecuencia)
```

`Counter` es una clase del módulo `collections` que cuenta la frecuencia de los elementos en una lista. Aquí, cuenta cuántas veces aparece cada número en la lista.

21. Crear un servidor HTTP básico

```
from http.server import SimpleHTTPRequestHandler, HTTPServer
server = HTTPServer(('localhost', 8080), SimpleHTTPRequestHandler)
print("Servidor iniciado en http://localhost:8080")
server.serve_forever()
```

Este código crea un servidor HTTP básico que sirve archivos desde el directorio actual en el puerto 8080. Se utiliza el módulo `http.server`, que viene integrado en Python.

22. Uso de `argparse` para argumentos en línea de comandos

```
import argparse
parser = argparse.ArgumentParser(description="Ejemplo de argparse")
parser.add_argument('numero', type=int, help="Número de entrada")
args = parser.parse_args()
print(f"El número ingresado es {args.numero}")
```

`argparse` es una biblioteca que facilita el manejo de argumentos desde la línea de comandos. Este código define un argumento `numero` que el usuario debe ingresar, y luego imprime el valor ingresado.

23. Hilos en Python

```
import threading
def imprimir_mensaje():
    print("Este es un mensaje desde otro hilo")
hilo = threading.Thread(target=imprimir_mensaje)
hilo.start()
```

```
hilo.join()
```

Este código crea un hilo usando la clase `Thread` del módulo `threading`. Los hilos permiten ejecutar varias tareas en paralelo. Aquí, el hilo imprime un mensaje desde una función separada.

24. Multiprocesamiento en Python

```
import multiprocessing
def proceso_func():
    print("Este es un proceso separado")
p = multiprocessing.Process(target=proceso_func)
p.start()
p.join()
```

El módulo `multiprocessing` permite ejecutar múltiples procesos en paralelo. En este ejemplo, se crea un proceso que ejecuta la función `proceso_func()` en paralelo.

25. Métodos mágicos `__str__` y `__repr__`

```
class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor
    def __str__(self):
        return f"Libro: {self.titulo} de {self.autor}"
    def __repr__(self):
        return f"Libro({self.titulo}, {self.autor})"
libro = Libro("1984", "George Orwell")
print(libro)
```

Este código define los métodos mágicos `__str__` y `__repr__`. `__str__` se usa para representar la cadena "amigable para humanos", mientras que `__repr__` es más técnica y orientada a los desarrolladores.

26. Ordenar objetos personalizados

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
```

```
        self.edad = edad
personas = [Persona("Ana", 30), Persona("Juan", 25)]
personas_ordenadas = sorted(personas, key=lambda p: p.edad)
for persona in personas_ordenadas:
    print(f"{persona.nombre}: {persona.edad}")
```

Aquí se define una clase `Persona` y se usa la función `sorted()` para ordenar una lista de objetos `Persona` según la edad. El argumento `key` permite especificar cómo ordenar los objetos.

27. Operaciones con conjuntos

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
union = conjunto1 | conjunto2
interseccion = conjunto1 & conjunto2
print(f"Unión: {union}, Intersección: {interseccion}")
```

Este código muestra cómo usar conjuntos en Python. El operador `|` calcula la unión de dos conjuntos, mientras que `&` calcula la intersección. Los conjuntos eliminan elementos duplicados y son útiles para operaciones matemáticas como estas.

28. Validación de entrada con `assert`

```
def dividir(a, b):
    assert b != 0, "El divisor no puede ser cero"
    return a / b
print(dividir(10, 2))
```

`assert` es una declaración utilizada para verificar si una condición es verdadera. En este ejemplo, se asegura que el divisor no sea cero antes de realizar la división. Si la condición es falsa, se lanza una excepción `AssertionError`.

29. Trabajar con `enumerate()`

```
frutas = ['manzana', 'naranja', 'uva']
for indice, fruta in enumerate(frutas):
    print(f"{indice}: {fruta}")
```

`enumerate()` es una función que permite iterar sobre una lista o secuencia, devolviendo tanto el índice como el valor del elemento. Este código imprime el índice y el nombre de cada fruta en la lista.

30. Graficar datos con `matplotlib`

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]
plt.plot(x, y)
plt.title("Ejemplo de gráfico")
plt.show()
```

Este código utiliza la biblioteca `matplotlib` para crear un gráfico de líneas. Se grafican los valores de `x` contra los valores de `y` y se muestra el gráfico en una ventana emergente con `plt.show()`.