Anexo. Lenguaje Python

python™

Gobierno de Canarias
Servicio Canario de Empleo
PIEC
PLAN INTEGRAL DE EMPLEO DE CANARIAS
master
SEPE

1. Introducción a Python

1.1 ¿Qué es Python?

Lenguaje de programación interpretado, de alto nivel y propósito general

- **Definición**: Python es un lenguaje de programación interpretado, lo que significa que el código se ejecuta directamente, sin necesidad de ser compilado primero. Es de alto nivel, lo que permite a los desarrolladores escribir programas de manera más intuitiva y sencilla comparado con lenguajes de bajo nivel.
- **Propósito general**: Python se puede usar para una amplia variedad de aplicaciones, desde desarrollo web hasta análisis de datos, inteligencia artificial, automatización de tareas, y por supuesto, seguridad informática.

Historia breve de Python

- **Creación**: Python fue creado por Guido van Rossum y su primera versión fue lanzada en 1991.
- **Evolución**: A lo largo de los años, Python ha evolucionado significativamente, con versiones importantes como Python 2.0 lanzada en el año 2000 y Python 3.0 en 2008. Python 3 introdujo muchas mejoras pero no fue completamente compatible con Python 2, lo que requirió a los desarrolladores adaptar su código.
- **Popularidad**: Actualmente, Python es uno de los lenguajes de programación más populares y ampliamente utilizados en el mundo debido a su simplicidad y versatilidad.

Usos comunes y su importancia en la seguridad informática

- **Usos comunes**: Python se utiliza en desarrollo web (Django, Flask), ciencia de datos (pandas, NumPy, SciPy), inteligencia artificial (TensorFlow, Keras), automatización, y más.
- **Seguridad informática**: En el ámbito de la seguridad informática, Python se utiliza para:
  - **Desarrollo de herramientas de análisis y pruebas de penetración**: Herramientas como Scapy para análisis de redes y Nmap para escaneo de puertos.
  - **Automatización de tareas**: Automatización de pruebas de seguridad, gestión de redes y sistemas.
  - **Criptografía**: Implementación de algoritmos de cifrado y hashing.

1.2 Instalación y Configuración

Instalación de Python en diferentes sistemas operativos

- **Windows**:
  1. Descarga el instalador de Python desde la página oficial python.org.
  2. Ejecuta el instalador y asegúrate de seleccionar la opción "Add Python to PATH".
  3. Completa la instalación siguiendo las indicaciones.
- **macOS**:
  1. Abre la Terminal.
  2. Usa brew (Homebrew) para instalar Python: brew install python.
  3. Verifica la instalación con python3 --version.
- **Linux**:
  1. Abre la Terminal.
  2. Usa el gestor de paquetes correspondiente (apt para Debian/Ubuntu, yum para Fedora, etc.):
     - Debian/Ubuntu: sudo apt update && sudo apt install python3.
     - Fedora: sudo dnf install python3.
  3. Verifica la instalación con python3 --version.

Uso de entornos virtuales (virtualenv)

- **Concepto**: Los entornos virtuales permiten aislar las dependencias de tu proyecto, evitando conflictos entre diferentes proyectos.
- **Instalación**:
  ```
  pip install virtualenv
  ```
- **Creación de un entorno virtual**:
  1. Navega a tu directorio de proyecto.
  2. Crea un entorno virtual:
     ```
     python3 -m venv env
     ```
  3. Activa el entorno virtual:
     - **Windows**: .\env\Scripts\activate
     - **macOS/Linux**: source env/bin/activate
  4. Para desactivar el entorno virtual: deactivate.

Instalación de un editor de texto o IDE

- **PyCharm**:
  1. Descarga PyCharm desde la página oficial jetbrains.com/pycharm.
  2. Sigue las instrucciones de instalación.
  3. Configura un nuevo proyecto y selecciona el intérprete de Python (puedes usar el entorno virtual creado previamente).
- **VSCode:**
  1. Descarga VSCode desde la página oficial code.visualstudio.com.
  2. Instala la extensión de Python:
     1. Abre VSCode.
     2. Ve a la sección de extensiones (Ctrl+Shift+X).
     3. Busca "Python" y selecciona la extensión de Microsoft.
     4. Haz clic en "Install".
  3. Configura el intérprete de Python para tu proyecto:
     1. Abre la paleta de comandos (Ctrl+Shift+P).
     2. Escribe Python: Select Interpreter y selecciona el intérprete (puedes usar el entorno virtual creado previamente).

2. Fundamentos de Python

2.1 Sintaxis Básica

Estructura de un programa en Python

Un programa en Python consiste en una secuencia de instrucciones que el intérprete ejecuta una por una. Aquí hay un ejemplo básico de un programa en Python que imprime "Hola, Mundo!" en la pantalla:

```
# HolaMundo.py
print("Hola, Mundo!")
```

Variables y tipos de datos

Las variables en Python no requieren una declaración explícita del tipo de dato. Puedes asignar cualquier tipo de dato a una variable, y Python manejará el tipo automáticamente.

Ejemplo:

```
# Variables y tipos de datos
x = 10          # Entero
y = 3.14        # Flotante
nombre = "Ana"  # Cadena
es_mayor = True # Booleano

print(x)
print(y)
print(nombre)
print(es_mayor)
```

Comentarios y uso adecuado

Los comentarios en Python comienzan con el símbolo # y se extienden hasta el final de la línea. Se usan para explicar el código y no se ejecutan.

Ejemplo:

```
# Este es un comentario en Python
x = 5  # Asignamos el valor 5 a la variable x
print(x)  # Imprimimos el valor de x
```

2.2 Operadores

Aritméticos

Los operadores aritméticos se usan para realizar operaciones matemáticas.

Ejemplo:

```
# Operadores aritméticos
a = 10
b = 3

print(a + b)  # Suma
print(a - b)  # Resta
print(a * b)  # Multiplicación
print(a / b)  # División
print(a % b)  # Módulo
print(a // b) # División entera
print(a ** b) # Exponenciación
```

Comparación

Los operadores de comparación se usan para comparar dos valores y devolver un valor booleano.

Ejemplo:

```
# Operadores de comparación
a = 10
b = 3
print(a == b)  # Igual a
print(a != b)  # No igual a
```

```
print(a > b)   # Mayor que
print(a < b)   # Menor que
print(a >= b)  # Mayor o igual a
print(a <= b)  # Menor o igual a
```

Lógicos

Los operadores lógicos se usan para combinar declaraciones condicionales.

Ejemplo:

```
# Operadores lógicos
x = True
y = False
print(x and y)  # AND lógico
print(x or y)   # OR lógico
print(not x)    # NOT lógico
```

Asignación

Los **operadores** de asignación se usan para asignar valores a las variables.

Ejemplo:

```
# Operadores de asignación
a = 10
a += 5  # a = a + 5
print(a)

b = 3
b *= 2  # b = b * 2
print(b)
```

2.3 Estructuras de Control

Condicionales (if, elif, else)

Las declaraciones condicionales se usan para ejecutar código basado en ciertas condiciones.

Ejemplo:

```
# Condicionales
edad = 18
if edad < 18:
    print("Eres menor de edad")
elif edad == 18:
    print("Tienes 18 años")
else:
    print("Eres mayor de edad")
```

Bucles (for, while)

Los bucles se usan para ejecutar un bloque de código varias veces.

Ejemplo con for:

```
# Bucle for
for i in range(5):  # range(5) genera los números 0, 1, 2, 3, 4
    print(i)
```

Ejemplo con while:

```
# Bucle while
contador = 0
while contador < 5:
```

```
    print(contador)
    contador += 1
```

3. Estructuras de Datos en Python

3.1 Listas

Creación, acceso, modificación

Las listas en Python son colecciones ordenadas que permiten almacenar múltiples elementos, que pueden ser de diferentes tipos.

Creación de una lista:

```
# Creación de listas
mi_lista = [1, 2, 3, 4, 5]
lista_mixta = [1, "Hola", 3.14, True]
```

Acceso a elementos de la lista:

```
# Acceso a elementos de la lista
print(mi_lista[0])  # Primer elemento (1)
print(mi_lista[2])  # Tercer elemento (3)
print(mi_lista[-1]) # Último elemento (5)
```

Modificación de elementos de la lista:

```
# Modificación de elementos de la lista
mi_lista[0] = 10
print(mi_lista)  # [10, 2, 3, 4, 5]
```

Métodos comunes

Python proporciona varios métodos para manipular listas.

Ejemplo de métodos comunes:

```
# Métodos comunes de listas
mi_lista = [1, 2, 3, 4, 5]
```

```
# Append
mi_lista.append(6)
print(mi_lista)  # [1, 2, 3, 4, 5, 6]
# Remove
mi_lista.remove(3)
print(mi_lista)  # [1, 2, 4, 5, 6]
# Sort
mi_lista.sort()
print(mi_lista)  # [1, 2, 4, 5, 6]
# Reverse
mi_lista.reverse()
print(mi_lista)  # [6, 5, 4, 2, 1]
# Index
print(mi_lista.index(4))  # 2
# Count
print(mi_lista.count(2))  # 1
```

Ejercicio 2: "Programa que ordene una lista de números"

Escribe un programa que tome una lista de números y la ordene en orden ascendente.

```
# Programa que ordene una lista de números
def ordenar_lista(numeros):
    numeros.sort()
    return numeros
mi_lista = [4, 2, 9, 1, 5, 6]
lista_ordenada = ordenar_lista(mi_lista)
print("Lista ordenada:", lista_ordenada)  # [1, 2, 4, 5, 6, 9]
```

3.2 Tuplas

Inmutabilidad y usos

Las tuplas son similares a las listas, pero son inmutables, lo que significa que no pueden ser modificadas una vez creadas.

Creación de una tupla:

```
# Creación de tuplas
mi_tupla = (1, 2, 3, 4, 5)
tupla_mixta = (1, "Hola", 3.14, True)
Intento de modificación (generará un error):
python
Copiar código
# Intento de modificación (esto generará un error)
# mi_tupla[0] = 10
```

Desempaquetado

El desempaquetado permite asignar los elementos de una tupla a variables individuales.

Ejemplo de desempaquetado:

```
# Desempaquetado de tuplas
mi_tupla = (1, 2, 3)
a, b, c = mi_tupla
print(a)  # 1
print(b)  # 2
print(c)  # 3
```

Ejercicio 3: "Uso de tuplas para almacenar coordenadas"

Escribe un programa que utilice tuplas para almacenar coordenadas (x, y) y las imprima.

```
# Uso de tuplas para almacenar coordenadas
def imprimir_coordenadas(coordenadas):
    for coord in coordenadas:
        print(f"Coordenada: x = {coord[0]}, y = {coord[1]}")

coordenadas = [(1, 2), (3, 4), (5, 6)]
imprimir_coordenadas(coordenadas)
# Salida:
# Coordenada: x = 1, y = 2
# Coordenada: x = 3, y = 4
# Coordenada: x = 5, y = 6
```

3.3 Diccionarios

Creación, acceso, modificación

Los diccionarios son colecciones desordenadas de pares clave-valor.

Creación de un diccionario:

```
# Creación de diccionarios
mi_diccionario = {"nombre": "Ana", "edad": 25, "ciudad": "Madrid"}
```

Acceso a valores en un diccionario:

```
# Acceso a valores en un diccionario
print(mi_diccionario["nombre"])  # Ana
print(mi_diccionario["edad"])    # 25
```

Modificación de valores en un diccionario:

```
# Modificación de valores en un diccionario
mi_diccionario["edad"] = 26
print(mi_diccionario)  # {'nombre': 'Ana', 'edad': 26, 'ciudad': 'Madrid'}
```

Métodos comunes

Ejemplo de métodos comunes:

```
# Métodos comunes de diccionarios
mi_diccionario = {"nombre": "Ana", "edad": 25, "ciudad": "Madrid"}

# Keys
print(mi_diccionario.keys())  # dict_keys(['nombre', 'edad', 'ciudad'])

# Values
print(mi_diccionario.values())  # dict_values(['Ana', 25, 'Madrid'])
```

```
# Items
print(mi_diccionario.items())  # dict_items([('nombre', 'Ana'), ('edad',
25), ('ciudad', 'Madrid')])

# Get
print(mi_diccionario.get("nombre"))  # Ana

# Pop
mi_diccionario.pop("edad")
print(mi_diccionario)  # {'nombre': 'Ana', 'ciudad': 'Madrid'}
```

Ejercicio 4: "Programa que cuente la frecuencia de palabras en un texto"

Escribe un programa que tome un texto como entrada y cuente la frecuencia de cada palabra.

```
# Programa que cuente la frecuencia de palabras en un texto
def contar_palabras(texto):
    palabras = texto.split()
    frecuencia = {}
    for palabra in palabras:
        if palabra in frecuencia:
            frecuencia[palabra] += 1
        else:
            frecuencia[palabra] = 1
    return frecuencia

texto = "hola mundo hola a todos"
frecuencia_palabras = contar_palabras(texto)
print(frecuencia_palabras)
# Salida: {'hola': 2, 'mundo': 1, 'a': 1, 'todos': 1}
```

4. Funciones

4.1 Definición y Uso de Funciones

Definición con def

En Python, las funciones se definen utilizando la palabra clave def, seguida del nombre de la función, paréntesis y dos puntos. El cuerpo de la función contiene el código que se ejecutará cuando se llame a la función.

Ejemplo básico:

```
# Definición de una función simple
def saludar():
    print("¡Hola, Mundo!")

# Llamada a la función
saludar()  # ¡Hola, Mundo!
```

Parámetros y retorno de valores

Las funciones pueden aceptar parámetros, que son valores que se pasan a la función para que realice operaciones con ellos. También pueden devolver valores usando la palabra clave return.

Ejemplo con parámetros y retorno de valores:

```
# Función con parámetros y retorno de valores
def sumar(a, b):
    return a + b
resultado = sumar(3, 5)
print("La suma es:", resultado)  # La suma es: 8
```

Ejercicio 5: "Función que verifique si una cadena es un palíndromo"

Escribe una función que verifique si una cadena es un palíndromo (se lee igual de adelante hacia atrás).

```
# Función que verifica si una cadena es un palíndromo
def es_palindromo(cadena):
    # Eliminar espacios y convertir a minúsculas
    cadena = cadena.replace(" ", "").lower()
    # Verificar si es igual al revés
    return cadena == cadena[::-1]

# Prueba de la función
cadena = "Anita lava la tina"
if es_palindromo(cadena):
    print(f'"{cadena}" es un palíndromo')
else:
    print(f'"{cadena}" no es un palíndromo')
```

4.2 Ámbito de las Variables

Variables locales y globales

Las variables definidas dentro de una función son locales y solo son accesibles dentro de esa función. Las variables definidas fuera de cualquier función son globales y pueden ser accesibles desde cualquier parte del código, aunque es buena práctica evitar modificar variables globales dentro de funciones.

Ejemplo de variables locales y globales:

```
# Variable global
x = 10

def mi_funcion():
    # Variable local
    y = 5
    print("Dentro de la función, x =", x)  # Acceso a variable global
    print("Dentro de la función, y =", y)

mi_funcion()
print("Fuera de la función, x =", x)
# print("Fuera de la función, y =", y)  # Esto generará un error porque
y es local
```

Ejercicio 6: "Funciones recursivas para calcular el factorial de un número"

Escribe una función recursiva que calcule el factorial de un número. El factorial de un número n se define como el producto de todos los números enteros positivos desde 1 hasta n (n!).

```
# Función recursiva para calcular el factorial de un número
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

# Prueba de la función
numero = 5
print(f"El factorial de {numero} es {factorial(numero)}")  # El factorial
de 5 es 120
```

5. Manejo de Archivos

5.1 Operaciones Básicas

Apertura, lectura, escritura y cierre de archivos

En Python, el manejo de archivos se realiza utilizando la función open(), que devuelve un objeto de archivo. Este objeto de archivo se puede utilizar para leer, escribir y cerrar archivos.

Apertura de archivos:

La función open() se usa para abrir un archivo. Acepta dos parámetros: el nombre del archivo y el modo en el que se desea abrir el archivo.

Modos de apertura:

- "r": Lectura (modo por defecto). El archivo debe existir.
- "w": Escritura. Si el archivo no existe, se crea. Si existe, se sobrescribe.
- "a": Añadir. Si el archivo no existe, se crea. Si existe, se añade al final.
- "b": Modo binario (usar en combinación con otros modos, por ejemplo, "rb" para lectura binaria).

Ejemplo de apertura de archivo:

```
# Apertura de un archivo en modo lectura
archivo = open("ejemplo.txt", "r")
```

Lectura de archivos:

Se puede leer el contenido de un archivo usando varios métodos:

- read(): Lee todo el contenido del archivo.

- readline(): Lee una línea del archivo.
- readlines(): Lee todas las líneas del archivo y las devuelve como una lista.

Ejemplo de lectura de archivo:

```
# Lectura de un archivo
archivo = open("ejemplo.txt", "r")
contenido = archivo.read()
print(contenido)
archivo.close()
```

Escritura en archivos:

Se puede escribir en un archivo usando los métodos write() y writelines().

Ejemplo de escritura en archivo:

```
# Escritura en un archivo
archivo = open("ejemplo.txt", "w")
archivo.write("Hola, este es un archivo de ejemplo.\n")
archivo.write("Esta es una nueva línea.")
archivo.close()
```

Cierre de archivos:

Es importante cerrar el archivo después de que se ha terminado de trabajar con él para liberar los recursos asociados. Esto se hace usando el método close().

Ejemplo de cierre de archivo:

```
archivo = open("ejemplo.txt", "r")
contenido = archivo.read()
print(contenido)
archivo.close()
```

Ejercicio 7: "Programa que lea un archivo de texto y muestre su contenido"

Escribe un programa que lea el contenido de un archivo de texto llamado texto.txt y muestre su contenido en la pantalla.

```
# Programa que lee un archivo de texto y muestra su contenido
def leer_archivo(nombre_archivo):
    try:
        with open(nombre_archivo, "r") as archivo:
            contenido = archivo.read()
            print(contenido)
    except FileNotFoundError:
        print(f"El archivo {nombre_archivo} no existe.")

# Llamada a la función con el nombre del archivo
leer_archivo("texto.txt")
```

5.2 Operaciones Avanzadas

Lectura y escritura con bloques with

El uso de with es una forma eficiente de manejar archivos, ya que asegura que el archivo se cierre correctamente después de que se ha terminado de trabajar con él, incluso si se produce una excepción.

Ejemplo de uso de with:

```
# Lectura con with
with open("ejemplo.txt", "r") as archivo:
    contenido = archivo.read()
    print(contenido)
# Escritura con with
with open("ejemplo.txt", "w") as archivo:
    archivo.write("Nuevo contenido del archivo.")
```

Lectura y escritura de archivos binarios

Para trabajar con archivos binarios (por ejemplo, imágenes o archivos ejecutables), se utiliza el modo binario ("b").

Ejemplo de lectura y escritura de archivos binarios:

```
# Lectura de archivo binario
with open("imagen.jpg", "rb") as archivo:
    contenido = archivo.read()
    print(contenido[:10])  # Imprime los primeros 10 bytes del archivo
# Escritura de archivo binario
with open("copia_imagen.jpg", "wb") as archivo:
    archivo.write(contenido)
```

Manipulación de archivos grandes

Para manejar archivos grandes, se pueden leer y procesar por partes en lugar de cargar todo el archivo en la memoria de una vez.

Ejemplo de lectura en bloques:

```
# Lectura de archivo grande en bloques
with open("archivo_grande.txt", "r") as archivo:
    while True:
        bloque = archivo.read(1024)  # Lee el archivo en bloques de
1024 bytes
        if not bloque:
            break
        print(bloque)
```

6. Programación Orientada a Objetos (POO)

6.1 Conceptos Básicos

Clases y objetos

En Python, una clase es un plano para crear objetos. Define un conjunto de atributos y métodos que caracterizan cualquier objeto de la clase. Un objeto es una instancia de una clase.

Ejemplo básico de clase y objeto:

```
# Definición de una clase
class Perro:
    # Método constructor
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    # Método de instancia
    def ladrar(self):
        print(f"{self.nombre} está ladrando")
# Creación de un objeto
mi_perro = Perro("Firulais", 3)

# Acceso a atributos y métodos
print(mi_perro.nombre)  # Firulais
print(mi_perro.edad)    # 3
mi_perro.ladrar()       # Firulais está ladrando
```

Atributos y métodos

Los atributos son variables que pertenecen a una clase. Los métodos son funciones que pertenecen a una clase y describen los comportamientos del objeto.

Ejemplo con atributos y métodos:

```
# Definición de una clase con atributos y métodos
class Coche:
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año

    def arrancar(self):
        print(f"El coche {self.marca} {self.modelo} ha arrancado.")

    def frenar(self):
        print(f"El coche {self.marca} {self.modelo} ha frenado.")

# Creación de objetos
mi_coche = Coche("Toyota", "Corolla", 2020)

# Uso de atributos y métodos
print(mi_coche.marca)  # Toyota
print(mi_coche.modelo) # Corolla
mi_coche.arrancar()    # El coche Toyota Corolla ha arrancado.
mi_coche.frenar()      # El coche Toyota Corolla ha frenado.
```

Ejercicio 8: "Creación de una clase para manejar cuentas bancarias"

Escribe una clase para manejar cuentas bancarias que incluya atributos para el saldo y métodos para depositar y retirar dinero.

```
# Clase para manejar cuentas bancarias
class CuentaBancaria:
    def __init__(self, titular, saldo=0):
        self.titular = titular
        self.saldo = saldo
    def depositar(self, cantidad):
        self.saldo += cantidad
        print(f"Depósito de {cantidad}. Nuevo saldo: {self.saldo}")
    def retirar(self, cantidad):
        if cantidad > self.saldo:
            print("Fondos insuficientes")
        else:
            self.saldo -= cantidad
            print(f"Retiro de {cantidad}. Nuevo saldo: {self.saldo}")

# Creación de una cuenta bancaria
mi_cuenta = CuentaBancaria("Juan Pérez", 1000)
# Prueba de métodos de la clase
mi_cuenta.depositar(500)   # Depósito de 500. Nuevo saldo: 1500
mi_cuenta.retirar(200)     # Retiro de 200. Nuevo saldo: 1300
mi_cuenta.retirar(2000)    # Fondos insuficientes
```

6.2 Herencia y Polimorfismo

Subclases y superclases

La herencia permite crear una nueva clase basada en una clase existente. La clase nueva se llama subclase, y la clase existente se llama superclase.

Ejemplo de herencia:

```
# Clase base o superclase
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre
    def hacer_sonido(self):
        pass
# Clase derivada o subclase
class Perro(Animal):
    def hacer_sonido(self):
        print(f"{self.nombre} dice: Guau")
class Gato(Animal):
    def hacer_sonido(self):
        print(f"{self.nombre} dice: Miau")

# Creación de objetos
mi_perro = Perro("Firulais")
mi_gato = Gato("Mishi")
# Uso de métodos
mi_perro.hacer_sonido()  # Firulais dice: Guau
mi_gato.hacer_sonido()   # Mishi dice: Miau
```

Ejercicio 9: "Clase derivada para cuentas de ahorro"

Escribe una clase derivada para manejar cuentas de ahorro que incluya un atributo para la tasa de interés y un método para aplicar intereses.

```
# Clase base
class CuentaBancaria:
    def __init__(self, titular, saldo=0):
        self.titular = titular
        self.saldo = saldo

    def depositar(self, cantidad):
        self.saldo += cantidad
        print(f"Depósito de {cantidad}. Nuevo saldo: {self.saldo}")

    def retirar(self, cantidad):
        if cantidad > self.saldo:
            print("Fondos insuficientes")
        else:
            self.saldo -= cantidad
            print(f"Retiro de {cantidad}. Nuevo saldo: {self.saldo}")

# Clase derivada
class CuentaAhorro(CuentaBancaria):
    def __init__(self, titular, saldo=0, tasa_interes=0.02):
        super().__init__(titular, saldo)
        self.tasa_interes = tasa_interes
```

```
    def aplicar_interes(self):
        interes = self.saldo * self.tasa_interes
        self.saldo += interes
        print(f"Interés aplicado: {interes}. Nuevo saldo: {self.saldo}")

# Creación de una cuenta de ahorro
mi_cuenta_ahorro = CuentaAhorro("María López", 2000, 0.03)

# Prueba de métodos de la clase derivada
mi_cuenta_ahorro.depositar(500)         # Depósito de 500. Nuevo saldo:
2500
mi_cuenta_ahorro.aplicar_interes()      # Interés aplicado: 75.0. Nuevo
saldo: 2575
mi_cuenta_ahorro.retirar(1000)        # Retiro de 1000. Nuevo saldo: 1575
```

7. Módulos y Paquetes

7.1 Importación de Módulos

Python permite organizar y reutilizar el código mediante módulos y paquetes. Un módulo es un archivo que contiene definiciones y declaraciones de Python, como funciones, clases y variables. Los paquetes son una forma de estructurar los módulos en directorios jerárquicos.

Módulos estándar

Python incluye una biblioteca estándar con muchos módulos útiles. Aquí se muestran algunos ejemplos de uso de los módulos math, os, y sys.

Ejemplo con el módulo math:

```
import math

# Uso de funciones del módulo math
print(math.sqrt(16))   # 4.0
print(math.pi)         # 3.141592653589793
print(math.factorial(5)) # 120
```

Ejemplo con el módulo os:

```
import os

# Uso de funciones del módulo os
print(os.name)                # nt (en Windows) o posix (en Unix)
print(os.getcwd())            # Muestra el directorio actual de trabajo
os.mkdir("nuevo_directorio")  # Crea un nuevo directorio
```

Ejemplo con el módulo sys:

```
import sys

# Uso de funciones del módulo sys
print(sys.version)            # Muestra la versión de Python
print(sys.argv)                 # Muestra los argumentos de la línea de
comandos
sys.exit()                    # Sale del programa
```

Creación e importación de módulos propios

Puedes crear tus propios módulos guardando el código Python en archivos .py y luego importarlos en otros archivos.

Ejemplo de creación de un módulo propio:

1. Crea un archivo mi_modulo.py con el siguiente contenido:

```
# mi_modulo.py
def saludar(nombre):
    print(f"Hola, {nombre}!")
def despedir(nombre):
    print(f"Adiós, {nombre}!")
```

2. Crea otro archivo main.py en el mismo directorio para importar y usar mi_modulo.py:

```
# main.py
import mi_modulo

mi_modulo.saludar("Juan")  # Hola, Juan!
mi_modulo.despedir("Juan") # Adiós, Juan!
```

Ejercicio 10: "Uso de módulos para dividir un programa en varios archivos"

Escribe un programa que esté dividido en varios archivos utilizando módulos. El programa debe incluir un archivo principal main.py y dos módulos, operaciones.py y mensajes.py.

1. **Crea el archivo operaciones.py:**

```
# operaciones.py
def sumar(a, b):
    return a + b

def restar(a, b):
    return a - b
```

2. **Crea el archivo mensajes.py:**

```
# mensajes.py
def mostrar_bienvenida():
    print("¡Bienvenido al programa de operaciones!")

def mostrar_despedida():
    print("¡Gracias por usar el programa!")
```

3. **Crea el archivo main.py:**

```
# main.py
import operaciones
import mensajes

mensajes.mostrar_bienvenida()
```

```
a = 10
b = 5

suma = operaciones.sumar(a, b)
resta = operaciones.restar(a, b)

print(f"La suma de {a} y {b} es: {suma}")
print(f"La resta de {a} y {b} es: {resta}")

mensajes.mostrar_despedida()
```

Al ejecutar main.py, el programa debe mostrar un mensaje de bienvenida, realizar las operaciones de suma y resta, mostrar los resultados y finalizar con un mensaje de despedida.

8. Seguridad Informática con Python

8.1 Introducción a la Seguridad Informática

Importancia de la seguridad en el desarrollo de software

La seguridad en el desarrollo de software es crucial para proteger la información y los recursos de una organización. Un software seguro ayuda a prevenir ataques cibernéticos, proteger la privacidad de los usuarios, y asegurar la integridad y disponibilidad de los datos.

En el desarrollo de software, se deben seguir buenas prácticas de seguridad para minimizar las vulnerabilidades y garantizar que el software sea resistente a las amenazas.

Principios básicos de seguridad informática

1. **Confidencialidad:** Garantiza que la información solo sea accesible para aquellas personas autorizadas.
2. **Integridad:** Asegura que la información no sea alterada de manera no autorizada.
3. **Disponibilidad:** Garantiza que la información y los recursos estén disponibles para su uso cuando se necesiten.
4. **Autenticidad:** Verifica la identidad de los usuarios y la fuente de los datos.
5. **No repudio:** Previene que una entidad niegue haber realizado una acción.

8.2 Ejemplos Prácticos

Hashing (hashlib)

El hashing es un proceso que toma una entrada y la convierte en una cadena de longitud fija. Es comúnmente utilizado para almacenar contraseñas de manera segura.

Ejemplo de hashing con hashlib:

```
import hashlib

def hash_contraseña(contraseña):
    # Crear un objeto hash SHA-256
    hash_obj = hashlib.sha256()
    # Actualizar el hash con la contraseña codificada
    hash_obj.update(contraseña.encode('utf-8'))
    # Obtener el hash en formato hexadecimal
    return hash_obj.hexdigest()

# Ejemplo de uso
contraseña = "mi_contraseña_segura"
hash_resultado = hash_contraseña(contraseña)
print(f"Hash de la contraseña: {hash_resultado}")
```

Encriptación y desencriptación (cryptography)

La encriptación es el proceso de convertir datos en un formato ilegible para proteger la información. La desencriptación convierte los datos encriptados de vuelta a su formato original.

Ejemplo de encriptación y desencriptación con cryptography:

```
from cryptography.fernet import Fernet

# Generar una clave
clave = Fernet.generate_key()
f = Fernet(clave)

def encriptar_mensaje(mensaje):
    return f.encrypt(mensaje.encode('utf-8'))

def desencriptar_mensaje(mensaje_encriptado):
    return f.decrypt(mensaje_encriptado).decode('utf-8')

# Ejemplo de uso
mensaje = "Este es un mensaje secreto"
mensaje_encriptado = encriptar_mensaje(mensaje)
print(f"Mensaje encriptado: {mensaje_encriptado}")

mensaje_desencriptado = desencriptar_mensaje(mensaje_encriptado)
print(f"Mensaje desencriptado: {mensaje_desencriptado}")
```

Ejercicio 11: "Programa para hash de contraseñas"

Escribe un programa que solicite al usuario una contraseña, la hashee utilizando SHA-256 y muestre el hash resultante.

```
import hashlib

def hash_contraseña(contraseña):
    hash_obj = hashlib.sha256()
    hash_obj.update(contraseña.encode('utf-8'))
    return hash_obj.hexdigest()

contraseña = input("Introduce una contraseña: ")
hash_resultado = hash_contraseña(contraseña)
print(f"Hash de la contraseña: {hash_resultado}")
```

Ejercicio 12: "Encriptación y desencriptación de un mensaje"

Escribe un programa que solicite al usuario un mensaje, lo encripte y luego lo desencripte utilizando la librería cryptography.

```
from cryptography.fernet import Fernet

# Generar una clave
clave = Fernet.generate_key()
f = Fernet(clave)

def encriptar_mensaje(mensaje):
    return f.encrypt(mensaje.encode('utf-8'))

def desencriptar_mensaje(mensaje_encriptado):
    return f.decrypt(mensaje_encriptado).decode('utf-8')

mensaje = input("Introduce un mensaje para encriptar: ")
mensaje_encriptado = encriptar_mensaje(mensaje)
print(f"Mensaje encriptado: {mensaje_encriptado}")

mensaje_desencriptado = desencriptar_mensaje(mensaje_encriptado)
print(f"Mensaje desencriptado: {mensaje_desencriptado}")
```

8.3 Análisis de Vulnerabilidades

Herramientas de análisis (scapy, nmap)

Python ofrece herramientas poderosas para el análisis de vulnerabilidades y el escaneo de redes. scapy y nmap son dos de las más utilizadas.

Ejemplo de uso de scapy para análisis de paquetes:

```
from scapy.all import *

# Escanear un paquete
def escanear_paquete(paquete):
    if paquete.haslayer(IP):
        ip_src = paquete[IP].src
        ip_dst = paquete[IP].dst
        print(f"IP de origen: {ip_src} -> IP de destino: {ip_dst}")

# Capturar paquetes
sniff(prn=escanear_paquete, count=10)
```

Ejemplo de uso de nmap para escaneo de puertos:

```
import nmap

# Inicializar el escáner
nm = nmap.PortScanner()

# Escanear el host
host = '127.0.0.1'
```

```
nm.scan(host, '22-443')

# Mostrar resultados
for host in nm.all_hosts():
    print(f"Host: {host}")
    for proto in nm[host].all_protocols():
        print(f"Protocolo: {proto}")
        lport = nm[host][proto].keys()
        for port in lport:
            print(f"Puerto: {port}\tEstado:
{nm[host][proto][port]['state']}")
```

Ejercicio 13: "Escaneo de puertos con nmap en Python"

Escribe un programa que realice un escaneo de puertos en un host especificado por el usuario utilizando nmap.

```
import nmap
def escanear_puertos(host):
    nm = nmap.PortScanner()
    nm.scan(host, '22-443')
    for proto in nm[host].all_protocols():
        lport = nm[host][proto].keys()
        for port in lport:
            print(f"Puerto: {port}\tEstado:
{nm[host][proto][port]['state']}")
host = input("Introduce la IP del host a escanear: ")
escanear_puertos(host)
```

9. Proyectos Finales

A continuación se presentan dos propuestas de proyectos que permitirán a los estudiantes aplicar los conceptos y técnicas de seguridad informática utilizando Python.

Propuesta de Proyecto 1: Sistema de gestión de usuarios con contraseñas seguras

Descripción:

Desarrollar un sistema de gestión de usuarios que permita registrar, autenticar y gestionar contraseñas de manera segura. El sistema debe incluir características como el hashing de contraseñas, almacenamiento seguro y verificación de contraseñas durante el inicio de sesión.

Requisitos del Proyecto:

1. **Registro de Usuarios:**
   - Solicitar nombre de usuario y contraseña.
   - Hashear la contraseña antes de almacenarla.
2. **Inicio de Sesión:**
   - Solicitar nombre de usuario y contraseña.
   - Verificar la contraseña hasheada contra la almacenada.
3. **Gestión de Contraseñas:**
   - Permitir a los usuarios cambiar su contraseña.
   - Utilizar un archivo o una base de datos para almacenar la información de los usuarios de manera segura.

Ejemplo de Implementación:

```
import hashlib
import json
import os

# Función para hashear contraseñas
def hash_contraseña(contraseña):
    hash_obj = hashlib.sha256()
    hash_obj.update(contraseña.encode('utf-8'))
    return hash_obj.hexdigest()

# Función para registrar un nuevo usuario
def registrar_usuario(nombre_usuario, contraseña):
    usuarios = cargar_usuarios()
    if nombre_usuario in usuarios:
        print("El usuario ya existe.")
        return False
    usuarios[nombre_usuario] = hash_contraseña(contraseña)
    guardar_usuarios(usuarios)
    print("Usuario registrado exitosamente.")
    return True

# Función para autenticar un usuario
def autenticar_usuario(nombre_usuario, contraseña):
    usuarios = cargar_usuarios()
    if nombre_usuario not in usuarios:
```

```
        print("Usuario no encontrado.")
        return False
    if usuarios[nombre_usuario] == hash_contraseña(contraseña):
        print("Inicio de sesión exitoso.")
        return True
    else:
        print("Contraseña incorrecta.")
        return False

# Función para cargar usuarios desde un archivo JSON
def cargar_usuarios():
    if not os.path.exists('usuarios.json'):
        return {}
    with open('usuarios.json', 'r') as archivo:
        return json.load(archivo)

# Función para guardar usuarios en un archivo JSON
def guardar_usuarios(usuarios):
    with open('usuarios.json', 'w') as archivo:
        json.dump(usuarios, archivo)

# Menú principal del sistema
def menu():
    while True:
        print("\nSistema de Gestión de Usuarios")
        print("1. Registrar Usuario")
```

```
        print("2. Iniciar Sesión")
        print("3. Salir")
        opcion = input("Seleccione una opción: ")

        if opcion == '1':
            nombre_usuario = input("Nombre de usuario: ")
            contraseña = input("Contraseña: ")
            registrar_usuario(nombre_usuario, contraseña)
        elif opcion == '2':
            nombre_usuario = input("Nombre de usuario: ")
            contraseña = input("Contraseña: ")
            autenticar_usuario(nombre_usuario, contraseña)
        elif opcion == '3':
            break
        else:
            print("Opción no válida. Intente de nuevo.")

# Ejecutar el menú principal
menu()
```

Propuesta de Proyecto 2: Herramienta de escaneo de redes personalizada

Descripción:

Desarrollar una herramienta de escaneo de redes que permita a los usuarios detectar dispositivos y puertos abiertos en una red local. La herramienta debe utilizar bibliotecas como scapy y nmap para realizar el escaneo.

Requisitos del Proyecto:

1. **Escaneo de Puertos:**
   - Escanear un rango de puertos en un host especificado.
   - Mostrar los puertos abiertos y sus servicios.
2. **Detección de Dispositivos:**
   - Detectar dispositivos conectados a la red local.
   - Mostrar las direcciones IP y MAC de los dispositivos detectados.
3. **Interfaz de Usuario:**
   - Proporcionar una interfaz de línea de comandos o gráfica para que los usuarios interactúen con la herramienta.

Ejemplo de Implementación:

```
import nmap
from scapy.all import ARP, Ether, srp

# Función para escanear puertos
def escanear_puertos(host):
    nm = nmap.PortScanner()
    nm.scan(host, '1-1024')
    print(f"Escaneo de puertos en {host}:")
    for proto in nm[host].all_protocols():
        lport = nm[host][proto].keys()
        for port in lport:
            print(f"Puerto: {port}\tEstado:
{nm[host][proto][port]['state']}")

# Función para detectar dispositivos en la red local
def detectar_dispositivos(red):
    arp = ARP(pdst=red)
    ether = Ether(dst="ff:ff:ff:ff:ff:ff")
    paquete = ether/arp
    result = srp(paquete, timeout=3, verbose=0)[0]
    dispositivos = []
    for sent, received in result:
        dispositivos.append({'ip': received.psrc, 'mac':
received.hwsrc})
    return dispositivos
```

```
# Menú principal de la herramienta
def menu():
    while True:
        print("\nHerramienta de Escaneo de Redes")
        print("1. Escanear Puertos")
        print("2. Detectar Dispositivos en la Red Local")
        print("3. Salir")
        opcion = input("Seleccione una opción: ")
        if opcion == '1':
            host = input("Introduce la IP del host a escanear: ")
            escanear_puertos(host)
        elif opcion == '2':
            red = input("Introduce el rango de la red (ej.
192.168.1.0/24): ")
            dispositivos = detectar_dispositivos(red)
            print("Dispositivos detectados:")
            for dispositivo in dispositivos:
                print(f"IP: {dispositivo['ip']}, MAC:
{dispositivo['mac']}")
        elif opcion == '3':
            break
        else:
            print("Opción no válida. Intente de nuevo.")
# Ejecutar el menú principal
menu()
```

10. Conclusiones y Recomendaciones

Repaso de los conceptos aprendidos

Al concluir el curso de Seguridad Informática en Python, es fundamental repasar y consolidar los conceptos y habilidades adquiridas. A continuación, se presenta un resumen de los principales temas tratados:

1. **Introducción a Python**
   - Comprender la historia y los usos de Python.
   - Instalación y configuración de Python y entornos de desarrollo.
2. **Fundamentos de Python**
   - Sintaxis básica, variables y tipos de datos.
   - Operadores y estructuras de control.
3. **Estructuras de Datos en Python**
   - Listas, tuplas y diccionarios.
   - Métodos y manipulaciones comunes.
4. **Funciones**
   - Definición y uso de funciones.
   - Ámbito de las variables y funciones recursivas.
5. **Manejo de Archivos**
   - Operaciones básicas (apertura, lectura, escritura y cierre de archivos).
   - Operaciones avanzadas con archivos.

6. **Programación Orientada a Objetos (POO)**
    - Conceptos básicos de clases y objetos.
    - Herencia y polimorfismo.
7. **Módulos y Paquetes**
    - Importación de módulos estándar y creación de módulos propios.
8. **Seguridad Informática con Python**
    - Principios básicos de seguridad informática.
    - Hashing, encriptación, análisis de vulnerabilidades y uso de herramientas de seguridad.
9. **Proyectos Finales**
    - Aplicación de los conocimientos en proyectos prácticos como sistemas de gestión de usuarios y herramientas de escaneo de redes.